PAU.

Programming Is Like Playing A Game.

JAVA Learning.

"The expert in anything was once a beginner."

Contents:

- Basic Type of java:

Pages::::

MD.ERSHADUL H. CHOUDHURY

B.Sc Engg (BUET), MS (Texas A&M University, USA), Life Fellow, IEB

Founder & Former Associate Professor CSE Dept. EWU

Chairperson & Associate Professor

Computer Science and Engineering Department

(CSE & CSIT Programs)

Primeasia University.

MEHJABIN RAHMAN

Lecturer in JAVA

Computer Science and Engineering Department

(CSE & CSIT Programs)

Primeasia University.

By MD: Murad Khan Ridoy

Date: 01/01/2019

Introduction java

Java is a high-level programming language originally developed by Sun Microsystems and released in 1995. Java runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX. This tutorial gives a complete understanding of Java. This reference will take you through simple and practical approaches while learning Java Programming language.

Overview

Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE]).

The latest release of the Java Standard Edition is Java SE 8. With the advancement of Java and its widespread popularity, multiple configurations were built to suit various types of platforms. For example: J2EE for Enterprise Applications, J2ME for Mobile Applications.

The new J2 versions were renamed as Java SE, Java EE, and Java ME respectively. Java is guaranteed to be **Write Once, Run Anywhere.**

Java is −

- **Object Oriented** − In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
- **Platform Independent** − Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.
- **Simple** − Java is designed to be easy to learn. If you understand the basic concept of OOP Java, it would be easy to master.
- **Secure** − With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
- **Architecture-neutral** − Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.
- **Portable** − Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset.

- **Robust** − Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
- **Multithreaded** − With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.
- **Interpreted** − Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.
- **High Performance** − With the use of Just-In-Time compilers, Java enables high performance.
- **Distributed** − Java is designed for the distributed environment of the internet.
- **Dynamic** − Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

Using Software

Tools You Will Need

For performing the examples discussed in this tutorial, you will need a Pentium 200-MHz computer with a minimum of 64 MB of RAM (128 MB of RAM recommended).

You will also need the following softwares −

- Linux 7.1 or Windows xp/7/8 operating system
- Java JDK 8
- Microsoft Notepad or any other text editor

This tutorial will provide the necessary skills to create GUI, networking, and web applications using Java.

Using compiler :

NetBeans

- **Developer:** Oracle
- **Platforms:** NetBeans is a cross platform IDE, supporting Windows, Mac and Linux
- **Pricing:** Free
- **Size:** 186 MB for the Java EE Version

IntelliJ IDEA

- **Developer:** JetBrains
- **Platforms:** OS X, Linux, Windows
- **Pricing:** Paid ($199 individual user license). Free for Students and open source projects.
- **Size:** 180 MB for the Community Edition, 290 MB for the Ultimate Edition

Eclipse

- **Developer:** Eclipse Foundation
- **Platforms:** Eclipse is available for Windows, Mac, Linux and OSX
- **Pricing:** Free
- **Size:** The Eclipse Luna download package for Java Developers is (almost) 155 MB.

Jdeveloper

- **Developer:** Oracle
- **Platforms:** Cross Platform
- **Pricing:** Free
- **Size:** Java Edition-181 MB, Studio Edition -1.8 GB

Dr. Java

- **Platforms:** Cross Platform
- **Pricing:** Free
- **Size:** 13 MB

BlueJ

- **Platforms:** Cross Platform
- **Pricing:** Free
- **Size:** 160-170 MB (including JDK)

jCreator

- **Developer:** Xinox Software
- **Platforms:** Windows only.
- **Pricing:** Free
- **Size:** 7.2 MB for Trial Version

jGrasp

- **Platforms:** Cross Platform
- **Pricing:** Free
- **Size:** 5MB

Greenfoot

- **Platforms:** Cross Platform
- **Pricing:** Free
- **Size:** 162 MB for Windows (JDK included)

Codenvy

- **Developer:** Codenvy Inc.
- **Platforms:** Cross Platform (Browser based)

- **Pricing:** Free for the community edition. Subscriptions for premium version starting from $1 per month.

Popular Java Editors

To write your Java programs, you will need a text editor. There are even more sophisticated IDEs available in the market. But for now, you can consider one of the following −

- **Notepad** − On Windows machine, you can use any simple text editor like Notepad (Recommended for this tutorial), TextPad.
- **Netbeans** − A Java IDE that is open-source and free which can be downloaded from https://www.netbeans.org/index.html.
- **Eclipse** − A Java IDE developed by the eclipse open-source community and can be downloaded from https://www.eclipse.org/.

Basic syntax

A Java program can be defined as a collection of objects that communicate by invoking each other's methods. Let us look at a simple java program and digg briefly into basic java concepts.

```
class HelloWorld
{
        public static void main(String[] args)
        {
                System.out.println ("Welcome to Hello World program");
        }
}
```

Class and Objects

A Class groups variables and operations together in coherent modules. A class can have fields, constructors and methods.Objects are instances of classes. When you create an object, that object is of a certain class. The class is like a template (or blueprint) telling how objects of that class should look..

Variables and DataTypes

Computer programs, read data from input devices, processes it and write it to screen, file or network.In a Java program data is kept in variables. Your Java program first declares the variables, then read data into them, execute operations on the variables, and then write the variables somewhere again.

Each variable has a data type. The data type determines what kind of data the variable can contain, and what operations you can execute on it.

Fields

A field is a variable that belongs to a class or an object.

Operations

In Java, Operations are the instructions you can run to process the data in variables.Some operations read and write the values of variables, while other operations controls the flow of program.

Methods

A Java method is a collection of java statements that are grouped together to perform an operation. When you call a method, it actually executes several statements in order. Methods are typically used when you need to group operations together, that you need to be able execute from several different places.

Constructors

Constructors are a special kind of java method that is executed when an object of that class is created. Constructors initializes the objects internal fields.

Interfaces

An interface is a programming structure that enforce certain properties on an object (class). For example, say we have a car class and a scooter class and a truck class. Each of these three classes should have a start_engine() action. How the "engine is started" for each vehicle is left to each particular class, but the fact that they must have a start_engine action is the domain of the interface.In its most common form, an interface is a group of related methods with empty bodies.

Packages

A package is a directory containing Java classes and interfaces. Packages groups related classes and interfaces, thus making modularization of your code easier.

Java Files

All Java code must reside inside a file with the extension .java . For instance, your first java class HelloWorld.java. An application can consist of many such .java files.

Syntax

Lets take a look at an example of .java file.

```
package corejavaguru;

import java.io.FileReader;

public class MyClass {

        protected final String fileName = "/opt/coreJava";
        static {
           //static class initializer
        }

   public MyClass() {

   }

   public static void main(String[] args) {

   }

   public String readFile() {
      //File read statements
   }
}
```

A typical Java file may contain the following elements:

- Package declaration
- Import statements
- Type declaration
- Fields declaration
- Class initializers
- Constructors
- Methods

Package Declaration

The first line in our .java file example shown above is the package declaration.

```
package corejavaguru;
```

The package declaration consists of the word package, a space, and then the name of the package the type is to be located in. The .java file should be located in a directory structure that matches the package name. Like for example, if package name is com.abc.xyz then .java file shouuld be in /com/abc/xyz folder.

Import Statements

Next line in our .java file is an import statement.

```
import java.io.FileReader;
```

An import statement tells the compiler which other Java files this current Java file is using. There can be multiple import statements in one java file.

Type Declaration

Type can be either a class, an interface, an enum.The type declaration is delimited by a { and a }.

```
public class MyClass {
}
```

Fields declaration

A Java field is a variable inside a class.The field declaration ends with a semicolon ;.

```
protected final String fileName = "/opt/coreJava";
```

Class initializers

Class initializers begins with a { and ends with a }. Inside this block you can put initialization code that is to be executed when a instance of the class is created. Class initializers can also be static. Then they are executed already when the class is loaded, and only once because the class is only loaded in the Java Virtual Machine once.The keyword static before the block

makes the class initializer block static.

```
static {
//static class initializer
}
```

Constructors

Constructors are a special kind of java method that is executed when an object of that class is created. Constructors initializes the objects internal fields.Constructors are similar to class initializers, except they can take parameters.

```
public MyClass() { }
```

Methods

A method is a group of Java statements that perform some operation on some data, and may or may not return a result.Java methods are where you put the operations on variables in your Java code. In other words, you group Java code inside Java methods. Java methods must be located inside a Java class.Java methods are similar to what is called functions or procedures in other programming languages.

```
public String readFile() {
//File read statements
}
```

A method can be static also.A static method belongs to the class, not objects of the class. That means that you can call a static method without having an object of the class the static method belongs to.

```
public static void main(String[] args) {
}
```

First Java Program:

Let us look at a simple code that would print the words *Simple Test Promgram*.

```
public class SimpleProgram {

  public static void main(String []args) {
    System.out.println("Simple Test Program");
  }
}
```

Lets look at how to save the file, compile and run the program. Please follow the steps given below:

- Open notepad and add the code as above.
- Save the file as : SimpleProgram.java.
- Open a command prompt window and go o the directory where you saved the class. Assume its D:\.

- Type ' javac SimpleProgram.java ' and press enter to compile your code. If there are no errors in your code the command prompt will take you to the next line.(Assumption : The path variable is set).
- Now type ' java SimpleProgram' to run your program.
- You will be able to see ' Simple Test Program ' printed on the window.

```
C : > javac SimpleProgram.java
C : > java SimpleProgram
Simple Test Program
```

Explain public static void main (String args[])…..

The **public** keyword is an access specifier, which allows the programmer to control the visibility of class members. When a class member is preceded by public, then that member may be accessed by code outside the class in which it is declared.In this case, main( ) must be declared as public, since it must be called by code outside of its class when the program is started. The keyword **static**allows main( ) to be called without having to instantiate a particular instance of the class. This is necessary since main( ) is called by the Java interpreter before any objects are made. The keyword **void** simply tells the compiler that main( ) does not return a value. As you will see, methods may also return values.
As stated, **main**( ) is the method called when a Java application begins. Keep in mind that Java is case-sensitive. Thus, Main is different from main. It is important to understand that the Java compiler will compile classes that do not contain a main( ) method. But the Java interpreter has no way to run these classes. So, if you had typed Main instead of main, the compiler would still compile your program. However, the Java interpreter would report an error because it would be unable to find the main( ) method.
Any information that you need to pass to a method is received by variables specified within the set of parentheses that follow the name of the method. These variables are called parameters. If there are no parameters required for a given method, you still need to include the empty parentheses. In main( ), there is only one parameter, albeit a complicated one. **String args[ ]** declares a parameter named args, which is an array of instances of the class String. Objects of type String store character strings. In this case, args receives any command-line arguments present when the program is executed.

Object and Classes

Java is an Object-Oriented Language. As a language that has the Object-Oriented feature, Java supports the following fundamental concepts −

- Polymorphism
- Inheritance
- Encapsulation
- Abstraction

- Classes
- Objects
- Instance
- Method
- Message Parsing

In this chapter, we will look into the concepts - Classes and Objects.

- **Object** − Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.
- **Class** − A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.

In this page, we will learn about Java objects and classes. In object-oriented programming technique, we design a program using objects and classes.

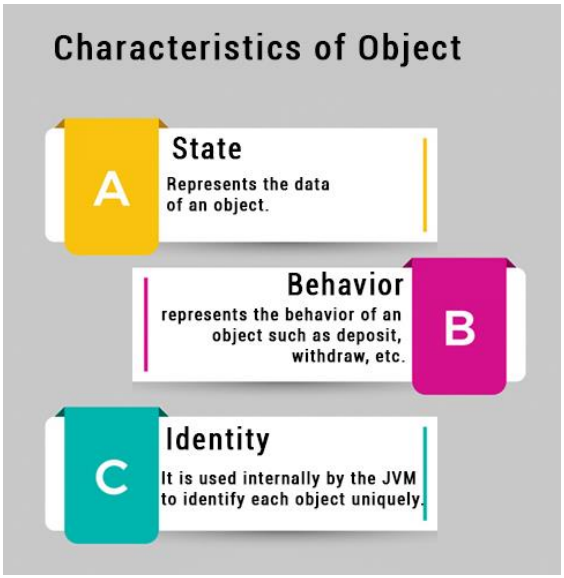
An object in Java is the physical as well as logical entity whereas a class in Java is a logical entity only.

What is an object in Java

An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

- **State:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.



For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

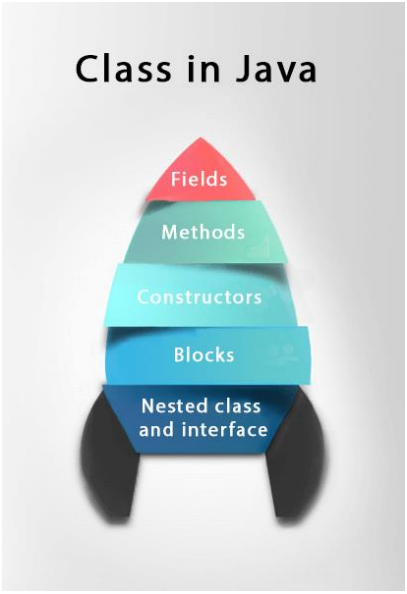
- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

What is a class in Java

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

- **Fields**
- **Methods**
- **Constructors**
- **Blocks**
- **Nested class and interface**



Syntax to declare a class:

```
1. class <class_name>{
2.     field;
3.     method;
4. }
```

Instance variable in Java

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

Method in Java

In Java, a method is like a function which is used to expose the behavior of an object.

Advantage of Method

- Code Reusability
- Code Optimization

---

new keyword in Java

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

Object and Class Example: main within the class

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

File: Student.java

```
1. //Java Program to illustrate how to define a class and fields
2. //Defining a Student class.
3. class Student{
4.  //defining fields
5.  int id;//field or data member or instance variable
6.  String name;
7.  //creating main method inside the Student class
8.  public static void main(String args[]){
9.   //Creating an object or instance
10.  Student s1=new Student();//creating an object of Student
11.  //Printing values of the object
12.  System.out.println(s1.id);//accessing member through reference variable
13.  System.out.println(s1.name);
14. }
15. }
```

Output:

```
0
null
```

Object and Class Example: main outside the class

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different java files or single java file. If you define multiple classes in a single java source file, it is a good idea to save the file name with the class name which has main() method.

File: TestStudent1.java

```
1. //Java Program to demonstrate having the main method in
2. //another class
3. //Creating Student class.
4. class Student{
```

```
5.  int id;
6.  String name;
7. }
8. //Creating another class TestStudent1 which contains the main method
9. class TestStudent1{
10. public static void main(String args[]){
11.  Student s1=new Student();
12.  System.out.println(s1.id);
13.  System.out.println(s1.name);
14. }
15. }
```

Output:

```
0
null
```

3 Ways to initialize object

There are 3 ways to initialize object in java.

1. By reference variable
2. By method
3. By constructor

1) Object and Class Example: Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

File: TestStudent2.java

```
1. class Student{
2.  int id;
3.  String name;
4. }
5. class TestStudent2{
6.  public static void main(String args[]){
7.   Student s1=new Student();
8.   s1.id=101;
9.   s1.name="Sonoo";
10.  System.out.println(s1.id+" "+s1.name);//printing members with a white space
11. }
12. }
```

Output:

```
101 Sonoo
```

We can also create multiple objects and store information in it through reference variable.

File: TestStudent3.java

```
1. class Student{
2.  int id;
3.  String name;
4. }
5. class TestStudent3{
6.  public static void main(String args[]){
7.   //Creating objects
8.   Student s1=new Student();
9.   Student s2=new Student();
10.  //Initializing objects
11.  s1.id=101;
12.  s1.name="Sonoo";
13.  s2.id=102;
14.  s2.name="Amit";
15.  //Printing data
16.  System.out.println(s1.id+" "+s1.name);
17.  System.out.println(s2.id+" "+s2.name);
18. }
19. }
```

Output:

```
101 Sonoo
102 Amit
```

2) Object and Class Example: Initialization through method

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

File: TestStudent4.java

```
1. class Student{
2.  int rollno;
3.  String name;
4.  void insertRecord(int r, String n){
5.   rollno=r;
6.   name=n;
7.  }
8.  void displayInformation(){System.out.println(rollno+" "+name);}
```

```
9.  }
10. class TestStudent4{
11.  public static void main(String args[]){
12.   Student s1=new Student();
13.   Student s2=new Student();
14.   s1.insertRecord(111,"Karan");
15.   s2.insertRecord(222,"Aryan");
16.   s1.displayInformation();
17.   s2.displayInformation();
18.  }
19. }
```

Output:

```
111 Karan
222 Aryan
```

As you can see in the above figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

---

3) Object and Class Example: Initialization through a constructor

We will learn about constructors in java later.

Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.

File: TestEmployee.java

```
1.  class Employee{
2.     int id;
3.     String name;
4.     float salary;
5.     void insert(int i, String n, float s) {
6.        id=i;
7.        name=n;
8.        salary=s;
9.     }
10.    void display(){System.out.println(id+" "+name+" "+salary);}
```

```
11. }
12. public class TestEmployee {
13. public static void main(String[] args) {
14.    Employee e1=new Employee();
15.    Employee e2=new Employee();
16.    Employee e3=new Employee();
17.    e1.insert(101,"ajeet",45000);
18.    e2.insert(102,"irfan",25000);
19.    e3.insert(103,"nakul",55000);
20.    e1.display();
21.    e2.display();
22.    e3.display();
23. }
24. }
```

Output:

```
101 ajeet 45000.0
102 irfan 25000.0
103 nakul 55000.0
```

Object and Class Example: Rectangle

There is given another example that maintains the records of Rectangle class.

File: TestRectangle1.java

```
1.  class Rectangle{
2.   int length;
3.   int width;
4.   void insert(int l, int w){
5.    length=l;
6.    width=w;
7.   }
8.   void calculateArea(){System.out.println(length*width);}
9.  }
10. class TestRectangle1{
11.  public static void main(String args[]){
12.   Rectangle r1=new Rectangle();
13.   Rectangle r2=new Rectangle();
14.   r1.insert(11,5);
15.   r2.insert(3,15);
16.   r1.calculateArea();
17.   r2.calculateArea();
18. }
19. }
```

Output:

```
55
45
```

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By deserialization
- By factory method etc.

We will learn these ways to create object later.

Anonymous object

Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only.

If you have to use an object only once, an anonymous object is a good approach. For example:

```
1. new Calculation();//anonymous object
```

Calling method through a reference:

```
1. Calculation c=new Calculation();
2. c.fact(5);
```

Calling method through an anonymous object

```
1. new Calculation().fact(5);
```

Let's see the full example of an anonymous object in Java.

```
1. class Calculation{
2.  void fact(int  n){
3.   int fact=1;
4.   for(int i=1;i<=n;i++){
5.    fact=fact*i;
6.   }
```

```
7.   System.out.println("factorial is "+fact);
8.  }
9.  public static void main(String args[]){
10.  new Calculation().fact(5);//calling method with anonymous object
11. }
12. }
```

Output:

```
Factorial is 120
```

Creating multiple objects by one type only

We can create multiple objects by one type only as we do in case of primitives.

Initialization of primitive variables:

```
1.  int a=10, b=20;
```

Initialization of refernce variables:

```
1.  Rectangle r1=new Rectangle(), r2=new Rectangle();//creating two objects
```

Let's see the example:

```
1.  //Java Program to illustrate the use of Rectangle class which
2.  //has length and width data members
3.  class Rectangle{
4.   int length;
5.   int width;
6.   void insert(int l,int w){
7.    length=l;
8.    width=w;
9.   }
10.  void calculateArea(){System.out.println(length*width);}
11. }
12. class TestRectangle2{
13.  public static void main(String args[]){
14.   Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
15.   r1.insert(11,5);
16.   r2.insert(3,15);
17.   r1.calculateArea();
18.   r2.calculateArea();
19. }
20. }
```

Output:

```
55
45
```

Real World Example: Account

File: TestAccount.java

```
1. //Java Program to demonstrate the working of a banking-system
2. //where we deposit and withdraw amount from our account.
3. //Creating an Account class which has deposit() and withdraw() methods
4. class Account{
5. int acc_no;
6. String name;
7. float amount;
8. //Method to initialize object
9. void insert(int a,String n,float amt){
10. acc_no=a;
11. name=n;
12. amount=amt;
13. }
14. //deposit method
15. void deposit(float amt){
16. amount=amount+amt;
17. System.out.println(amt+" deposited");
18. }
19. //withdraw method
20. void withdraw(float amt){
21. if(amount<amt){
22. System.out.println("Insufficient Balance");
23. }else{
24. amount=amount-amt;
25. System.out.println(amt+" withdrawn");
26. }
27. }
28. //method to check the balance of the account
29. void checkBalance(){System.out.println("Balance is: "+amount);}
30. //method to display the values of an object
31. void display(){System.out.println(acc_no+" "+name+" "+amount);}
32. }
33. //Creating a test class to deposit and withdraw amount
34. class TestAccount{
35. public static void main(String[] args){
36. Account a1=new Account();
37. a1.insert(832345,"Ankit",1000);
38. a1.display();
39. a1.checkBalance();
40. a1.deposit(40000);
```

```
41. a1.checkBalance();
42. a1.withdraw(15000);
43. a1.checkBalance();
44. }}
```

Output:

```
832345 Ankit 1000.0
Balance is: 1000.0
40000.0 deposited
Balance is: 41000.0
15000.0 withdrawn
Balance is: 26000.0
```

Constructors in Java

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the object is created, and memory is allocated for the object.

It is a special type of method which is used to initialize the object.

Calling a Constructor

You call a constructor when you create a new instance of the class containing the constructor. Here is a Java constructor call example:

```
MyClass myClassVar = new MyClass();
```

This example invokes (calls) the no-argument constructor for MyClass as defined earlier in this text.

In case you want to pass parameters to the constructor, you include the parameters between the parentheses after the class name, like this:

```
MyClass myClassVar = new MyClass(1975);
```

This example passes one parameter to the MyClass constructor that takes an int as parameter.

Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Rules for creating Java constructor

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

Note: We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

```
1. <class_name>(){}
```

Example of default constructor

In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
1. //Java Program to create and call a default constructor
2. class Bike1{
3. //creating a default constructor
4. Bike1(){System.out.println("Bike is created");}
5. //main method
```

```
6.  public static void main(String args[]){
7.  //calling a default constructor
8.  Bike1 b=new Bike1();
9.  }
10. }
```

Output:

```
Bike is created
```

> *Rule: If there is no constructor in a class, compiler automatically creates a default constructor.*

Example of default constructor that displays the default values

```
1.  //Let us see another example of default constructor
2.  //which displays the default values
3.  class Student3{
4.  int id;
5.  String name;
6.  //method to display the value of id and name
7.  void display(){System.out.println(id+" "+name);}
8.
9.  public static void main(String args[]){
10. //creating objects
11. Student3 s1=new Student3();
12. Student3 s2=new Student3();
13. //displaying values of the object
14. s1.display();
15. s2.display();
16. }
17. }
```

Output:

```
0 null
0 null
```

Explanation: In the above class,you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

Java Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor.

Why use the parameterized constructor?

The parameterized constructor is used to provide different values to the distinct objects. However, you can provide the same values also.

Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

```
1.  //Java Program to demonstrate the use of parameterized constructor
2.  class Student4{
3.     int id;
4.     String name;
5.     //creating a parameterized constructor
6.     Student4(int i,String n){
7.     id = i;
8.     name = n;
9.     }
10.    //method to display the values
11.    void display(){System.out.println(id+" "+name);}
12.
13.    public static void main(String args[]){
14.    //creating objects and passing values
15.    Student4 s1 = new Student4(111,"Karan");
16.    Student4 s2 = new Student4(222,"Aryan");
17.    //calling method to display the values of object
18.    s1.display();
19.    s2.display();
20.   }
21. }
```

Output:

```
111 Karan
222 Aryan
```

Constructor Overloading in Java

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

Example of Constructor Overloading

```
1. //Java program to overload constructors in java
2. class Student5{
3.    int id;
4.    String name;
5.    int age;
6.    //creating two arg constructor
7.    Student5(int i,String n){
8.    id = i;
9.    name = n;
10.   }
11.   //creating three arg constructor
12.   Student5(int i,String n,int a){
13.   id = i;
14.   name = n;
15.   age=a;
16.   }
17.   void display(){System.out.println(id+" "+name+" "+age);}
18.
19.   public static void main(String args[]){
20.   Student5 s1 = new Student5(111,"Karan");
21.   Student5 s2 = new Student5(222,"Aryan",25);
22.   s1.display();
23.   s2.display();
24.  }
25. }
```

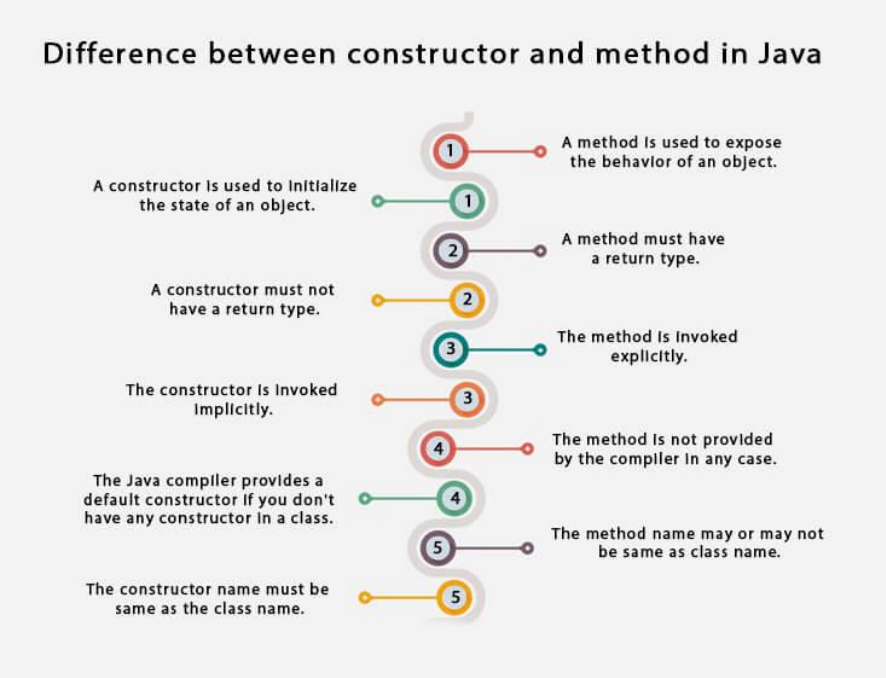
Output:

```
111 Karan 0
222 Aryan 25
```

Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as class name.



Java Copy Constructor

There is no copy constructor in java. However, we can copy the values from one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

In this example, we are going to copy the values of one object into another using java constructor.

```
1. //Java program to initialize the values from one object to another
2. class Student6{
3.    int id;
4.    String name;
5.    //constructor to initialize integer and string
6.    Student6(int i,String n){
7.    id = i;
8.    name = n;
9.    }
10.   //constructor to initialize another object
11.   Student6(Student6 s){
```

```
12.    id = s.id;
13.    name =s.name;
14.    }
15.    void display(){System.out.println(id+" "+name);}
16.
17.    public static void main(String args[]){
18.    Student6 s1 = new Student6(111,"Karan");
19.    Student6 s2 = new Student6(s1);
20.    s1.display();
21.    s2.display();
22.   }
23. }
```

Output:

```
111 Karan
111 Karan
```

Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```
1.  class Student7{
2.     int id;
3.     String name;
4.     Student7(int i,String n){
5.     id = i;
6.     name = n;
7.     }
8.     Student7(){}
9.     void display(){System.out.println(id+" "+name);}
10.
11.    public static void main(String args[]){
12.    Student7 s1 = new Student7(111,"Karan");
13.    Student7 s2 = new Student7();
14.    s2.id=s1.id;
15.    s2.name=s1.name;
16.    s1.display();
17.    s2.display();
18.   }
19. }
```

Output:

```
111 Karan
111 Karan
```

Basic Data types

There are two data types available in Java −

- Primitive Data Types
- Reference/Object Data Types

Primitive Data Types

There are eight primitive datatypes supported by Java. Primitive datatypes are predefined by the language and named by a keyword. Let us now look into the eight primitive data types in detail.

byte

- Byte data type is an 8-bit signed two's complement integer
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive)(2^7 -1)
- Default value is 0
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an integer.
- Example: byte a = 100, byte b = -50

short

- Short data type is a 16-bit signed two's complement integer
- Minimum value is -32,768 (-2^15)
- Maximum value is 32,767 (inclusive) (2^15 -1)
- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an integer
- Default value is 0.
- Example: short s = 10000, short r = -20000

int

- Int data type is a 32-bit signed two's complement integer.
- Minimum value is - 2,147,483,648 (-2^31)
- Maximum value is 2,147,483,647(inclusive) (2^31 -1)
- Integer is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0
- Example: int a = 100000, int b = -200000

long

- Long data type is a 64-bit signed two's complement integer
- Minimum value is -9,223,372,036,854,775,808(-2^63)
- Maximum value is 9,223,372,036,854,775,807 (inclusive)(2^63 -1)
- This type is used when a wider range than int is needed
- Default value is 0L
- Example: long a = 100000L, long b = -200000L

float

- Float data type is a single-precision 32-bit IEEE 754 floating point
- Float is mainly used to save memory in large arrays of floating point numbers
- Default value is 0.0f
- Float data type is never used for precise values such as currency
- Example: float f1 = 234.5f

double

- double data type is a double-precision 64-bit IEEE 754 floating point
- This data type is generally used as the default data type for decimal values, generally the default choice
- Double data type should never be used for precise values such as currency
- Default value is 0.0d
- Example: double d1 = 123.4

boolean

- boolean data type represents one bit of information
- There are only two possible values: true and false

- This data type is used for simple flags that track true/false conditions
- Default value is false
- Example: boolean one = true

char

- char data type is a single 16-bit Unicode character
- Minimum value is '\u0000' (or 0)
- Maximum value is '\uffff' (or 65,535 inclusive)
- Char data type is used to store any character
- Example: char letterA = 'A'

Reference Datatypes

- Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy, etc.
- Class objects and various type of array variables come under reference datatype.
- Default value of any reference variable is null.
- A reference variable can be used to refer any object of the declared type or any compatible type.
- Example: Animal animal = new Animal("giraffe");

Java Literals

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.

Literals can be assigned to any primitive type variable. For example −

```
byte a = 68;

char a = 'A';
```

byte, int, long, and short can be expressed in decimal(base 10), hexadecimal(base 16) or octal(base 8) number systems as well.

Prefix 0 is used to indicate octal, and prefix 0x indicates hexadecimal when using these number systems for literals. For example −

```
int decimal = 100;

int octal = 0144;

int hexa =  0x64;
```

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are –

Example

```
"Hello World"

"two\nlines"

"\"This is in quotes\""
```

String and char types of literals can contain any Unicode characters. For example –

```
char a = '\u0001';

String a = "\u0001";
```

Java language supports few special escape sequences for String and char literals as well. They are –

Notation	Character represented
\n	Newline (0x0a)
\r	Carriage return (0x0d)
\f	Formfeed (0x0c)
\b	Backspace (0x08)
\s	Space (0x20)
\t	Tab
\"	Double quote
\'	Single quote
\\	backslash

\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)

Variable Types

A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

You must declare all variables before they can be used. Following is the basic form of a variable declaration −

```
data type variable [ = value][, variable [ = value] ...] ;
```

Here *data type* is one of Java's datatypes and *variable* is the name of the variable. To declare more than one variable of the specified type, you can use a comma-separated list.

Following are valid examples of variable declaration and initialization in Java −

Example

```
int a, b, c;         // Declares three ints, a, b, and c.
int a = 10, b = 10;  // Example of initialization
byte B = 22;         // initializes a byte type variable B.
double pi = 3.14159; // declares and assigns a value of PI.
char a = 'a';        // the char variable a iis initialized with value 'a'
```

This chapter will explain various variable types available in Java Language. There are three kinds of variables in Java −

- Local variables
- Instance variables
- Class/Static variables

Local Variables

- Local variables are declared in methods, constructors, or blocks.

- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor, or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables, so local variables should be declared and an initial value should be assigned before the first use.

Example

Here, *age* is a local variable. This is defined inside *pupAge()* method and its scope is limited to only this method.

```
public class Test {

  public void pupAge() {

    int age = 0;

    age = age + 7;

    System.out.println("Puppy age is : " + age);

  }


  public static void main(String args[]) {

    Test test = new Test();

    test.pupAge();

  }

}
```

This will produce the following result −

Output

```
Puppy age is: 7
```

Example

Following example uses *age* without initializing it, so it would give an error at the time of compilation.

```
public class Test {

  public void pupAge() {
```

```
    int age;

    age = age + 7;

    System.out.println("Puppy age is : " + age);

  }


  public static void main(String args[]) {

    Test test = new Test();

    test.pupAge();

  }
}
```

This will produce the following error while compiling it –

Output

```
Test.java:4:variable number might not have been initialized
age = age + 7;
      ^
1 error
```

Instance Variables

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However, visibility for subclasses can be given for these variables with the use of access modifiers.

- Instance variables have default values. For numbers, the default value is 0, for Booleans it is false, and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class. However, within static methods (when instance variables are given accessibility), they should be called using the fully qualified name. *ObjectReference.VariableName*.

Example

```
import java.io.*;
public class Employee {

  // this instance variable is visible for any child class.
  public String name;

  // salary  variable is visible in Employee class only.
  private double salary;

  // The name variable is assigned in the constructor.
  public Employee (String empName) {
    name = empName;
  }

  // The salary variable is assigned a value.
  public void setSalary(double empSal) {
    salary = empSal;
  }

  // This method prints the employee details.
  public void printEmp() {
    System.out.println("name  : " + name );
```

```
    System.out.println("salary :" + salary);
  }

  public static void main(String args[]) {
    Employee empOne = new Employee("Ransika");
    empOne.setSalary(1000);
    empOne.printEmp();
  }
}
```

This will produce the following result −

Output

```
name  : Ransika
salary :1000.0
```

Class/Static Variables

- Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final, and static. Constant variables never change from their initial value.
- Static variables are stored in the static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally, values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name *ClassName.VariableName*.

- When declaring class variables as public static final, then variable names (constants) are all in upper case. If the static variables are not public and final, the naming syntax is the same as instance and local variables.

Example

```
import java.io.*;

public class Employee {


  // salary  variable is a private static variable

  private static double salary;


  // DEPARTMENT is a constant

  public static final String DEPARTMENT = "Development ";


  public static void main(String args[]) {

    salary = 1000;

    System.out.println(DEPARTMENT + "average salary:" + salary);

  }

}
```

This will produce the following result –

Output

```
Development average salary:1000
```

Java Operators

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators

- Assignment Operators

Arithmetic Operators

Arithmetic operators are used in mathematical like addition, subtraction etc. The following table lists the arithmetic operators:

Assume that int X = 10 and int Y = 20

Operators	Description
+	Addition – Adds values on either side of the operator
-	Subtraction – Subtracts right hand operand from left hand operand
*	Multiplication – Multiplies values on either side of the operand
/	Division - Divides left hand operand by right hand operand
%	Modulus - Divides left hand operand by right hand operand and returns remainder
++	Increment - Increase the value of operand by 1
--	Decrement - Decrease the value of operand by 1

Example:

```
public class Main{
        public static void main(String args[]{
                Int X = 10;
                Int Y = 20;
                System.out.println("Addition (X+Y) = "+(X+Y)); // return 30
                System.out.println("Subtraction  (X-Y) = "+(X-Y)); // return -10
                System.out.println("Multiplication (X*Y) = "+(X*Y)); // return 200
                System.out.println("Division (Y/X) = "+(Y/X)); // return 2
                System.out.println("Addition (Y%X) = "+(Y%X)); // return 0
                Y++;
                System.out.println("Increment Y = "+Y); // return 21
                X--;
                System.out.println("Decrement X = "+X); // return 9
        }
}
```

Relational Operators

There are following relational operators supported by Java language like ==, ! = etc. Assume variable X=10 and variable Y=20 then:

Operator	Description
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right

Operator	Description
	operand, if yes then condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

Example :

```
public class Main{
                public static void main(String args[]){
                int X = 10;
                int Y = 20;
                System.out.println("(X == Y) = "+(X == Y));
                System.out.println("(X != Y) = "+(X != Y));
                System.out.println("(X > Y) = "+(X > Y));
                System.out.println("(X < Y) = "+(X < Y));
                System.out.println("(X >= Y) = "+(X >= Y));
                System.out.println("(X <= Y) = "+(X <= Y));
        }
}
```

Bitwise Operators

Java defines several bitwise operators like &, | etc which can be applied to the integer types(long, int, short, char, and byte).

Bitwise operator works on bits(0 or 1) and perform bit by bit operation. Assume if x = 60; and y = 13; Now in binary format they will be as follows:

x = 0011 1100
y = 0000 1101
---
x&y = 0000 1100
x|y = 0011 1101
x^y = 0011 0001
~x = 1100 0011

The following table lists the bitwise operators:
Assume integer variable X=60 and variable Y=13 then:

Operator	Description
&	Binary AND Operator copies a bit to the result if it exists in both operands.
|	Binary OR Operator copies a bit if it exists in eather operand.
^	Binary XOR Operator copies the bit if it is set in one operand but not both.
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.
<<	Binary Left Shift Operator. The left operands value is moved left by

Operator	Description
>>	the number of bits specified by the right operand. Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.
>>>	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.

Example :

```
public class Main{
        public static void main(String args[]){
                int X = 60;
                int Y = 13;
                System.out.println("(X & Y) = "+(X & Y));
                System.out.println("(X | Y) = "+(X | Y));
                System.out.println("(X ^ Y) = "+(X ^ Y));
                System.out.println("(~X) = "+(~X));
                System.out.println("(X << Y) = "+(X << 2));
                System.out.println("(X >> Y) = "+(X >> 3));
                System.out.println("(X >>> Y) = "+(X >>> 1));
        }
}
```

Logical Operators

The following table lists the logical operators like &&, || etc. This logical operator use for join two condition.

Assume boolean variables X=true and variable Y=false then:

Operator	Description
&&	Called Logical AND operator. If both the operands are non zero then then condition becomes true.
||	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.

Example:

```
public class Main{
        public static void main(String args[]){
                int X = 60;
                int Y = 13;
                if((X == Y) && (X != Y)){
                        System.out.println("True");
```

```
                }else{
                        System.out.println("False");
                }
                if((X == Y) || (X != Y)){
                        System.out.println("True");
                }
                else{
                        System.out.println("False");
                }
        }
}
```

Assignment Operators

There are following assignment operators supported by Java language:

Operator	Description
=	Simple assignment operator, Assigns values from right side operands to left side operand
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand
<<=	Left shift AND assignment operator
>>=	Right shift AND assignment operator
&=	Bitwise AND assignment operator
^=	Bitwise exclusive OR and assignment operator
|=	Bitwise inclusive OR and assignment operator

Example:

```
public class Main{
     public static void main(String args[]){
                int X = 60;
                int Y = 13;
                X += 1;
                System.out.println("X+=1 : "+X);
                Y<<=1;
                System.out.println("Y<<=1 : "+Y);
                /* Return 26 : 13(binary - 00001101) shift one bit left means
26(00011010) */
```

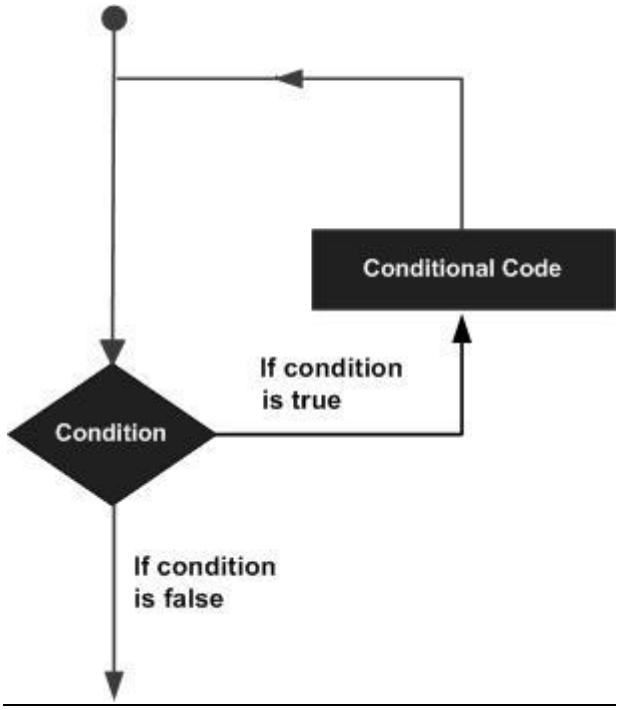
```
        }
}
```

Loop Control

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A **loop** statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages −



while loop

A **while** loop statement in Java programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

The syntax of a while loop is −

```
while(Boolean_expression) {
```

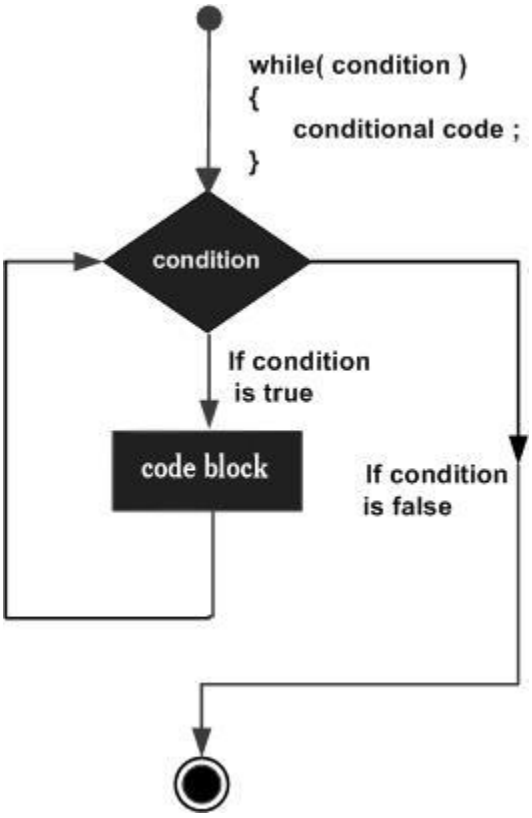
```
  // Statements
}
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non zero value.

When executing, if the *boolean_expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.

When the condition becomes false, program control passes to the line immediately following the loop.

Flow Diagram



Here, key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```
public class Test {
```

```
  public static void main(String args[]) {
    int x = 10;

    while( x < 20 ) {
      System.out.print("value of x : " + x );
      x++;
      System.out.print("\n");
    }
  }
}
```

This will produce the following result −

Output

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

for loop

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times.

A **for** loop is useful when you know how many times a task is to be repeated.

Syntax

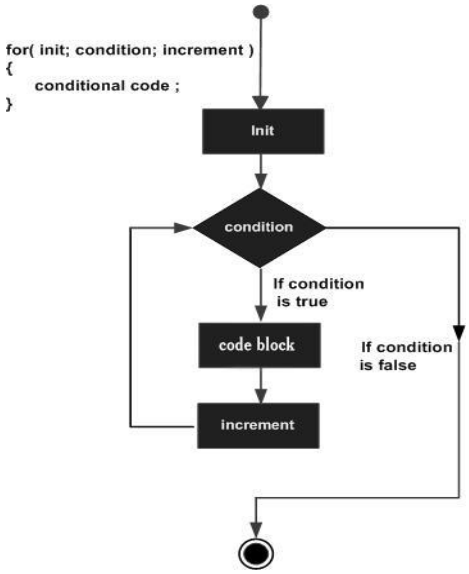
The syntax of a for loop is −

```
for(initialization; Boolean_expression; update) {
  // Statements
}
```

Here is the flow of control in a **for** loop −

- The **initialization** step is executed first, and only once. This step allows you to declare and initialize any loop control variables and this step ends with a semi colon (;).
- Next, the **Boolean expression** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop will not be executed and control jumps to the next statement past the for loop.
- After the **body** of the for loop gets executed, the control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank with a semicolon at the end.
- The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

Flow Diagram



Example

Following is an example code of the for loop in Java.

```
public class Test {

  public static void main(String args[]) {

    for(int x = 10; x < 20; x = x + 1) {
      System.out.print("value of x : " + x );
      System.out.print("\n");
```

```
    }
  }
}
```

This will produce the following result −

Output

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

do...while

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax

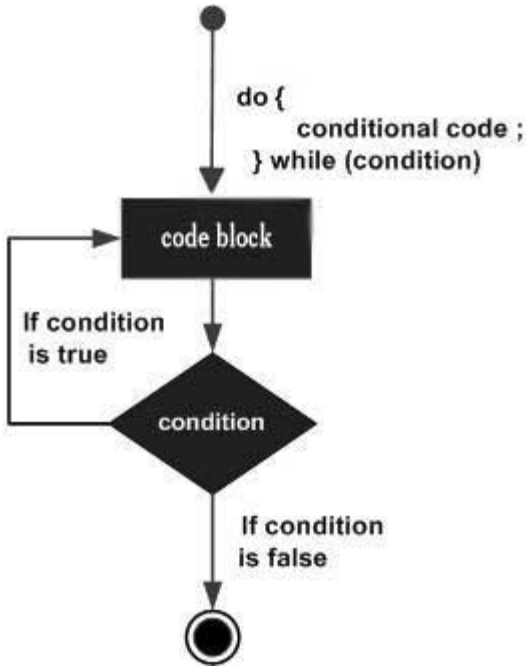
Following is the syntax of a do...while loop −

```
do {
  // Statements
}while(Boolean_expression);
```

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.

If the Boolean expression is true, the control jumps back up to do statement, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

Flow Diagram



Example

```
public class Test {


  public static void main(String args[]) {
    int x = 10;


    do {
      System.out.print("value of x : " + x );
      x++;
      System.out.print("\n");
    }while( x < 20 );
  }
}
```

This will produce the following result −

Output

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

break statement

The **break** statement in Java programming language has the following two usages −

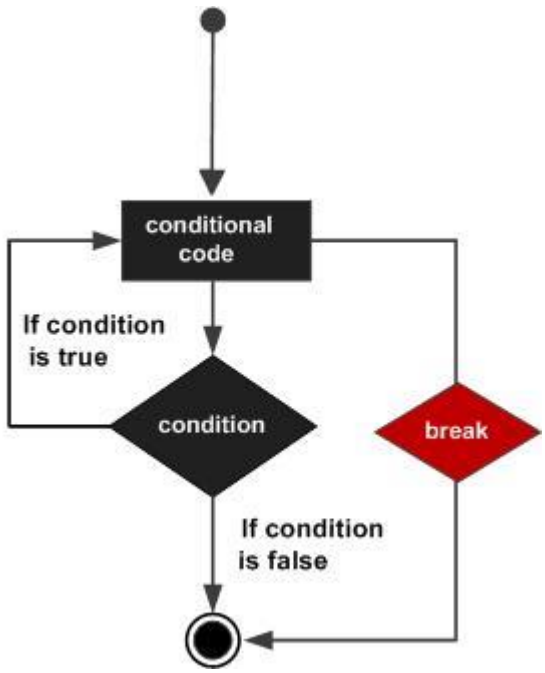
- When the **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- It can be used to terminate a case in the **switch** statement (covered in the next chapter).

Syntax

The syntax of a break is a single statement inside any loop −

```
break;
```

Flow Diagram



Example

```
public class Test {


  public static void main(String args[]) {
    int [] numbers = {10, 20, 30, 40, 50};


    for(int x : numbers ) {
      if( x == 30 ) {
        break;
      }
      System.out.print( x );
      System.out.print("\n");
    }
  }
}
```

This will produce the following result −

Output

```
10
20
```

continue

The **continue** keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

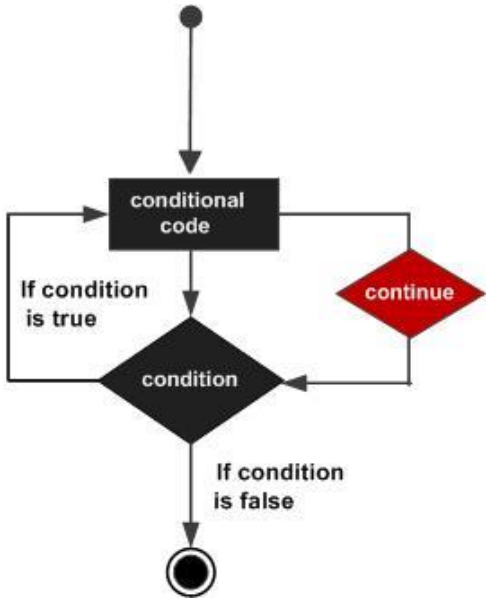
- In a for loop, the continue keyword causes control to immediately jump to the update statement.
- In a while loop or do/while loop, control immediately jumps to the Boolean expression.

Syntax

The syntax of a continue is a single statement inside any loop −

```
continue;
```

Flow Diagram



Example

```
public class Test {


  public static void main(String args[]) {
    int [] numbers = {10, 20, 30, 40, 50};


    for(int x : numbers ) {
      if( x == 30 ) {
        continue;
      }
      System.out.print( x );
      System.out.print("\n");
    }
  }
}
```

This will produce the following result −

Output

```
10
20
40
50
```

Decision Making(if/else)

If statement:

The if statement is Java's conditional branch statement.

Syntax:

```
If(Boolean expression){
// Code here
}
```

Example:

```
public class Main{
        public static void main(String args[]){
                int a = 10;
                int b = 20;
                if(a>b){
                        System.out.println("a is greater than b");
                }
                if(b<a){
                        System.out.println("b is less than a");
                }
}
}
```

Output:

a is greater than b

b is less than a

if else :

Syntax:

```
if(Boolean condition){
        // statement1
}else{
        // statement2
}
```

This if else work like : if condition is true than statement1 is executed otherwise statement2 is executed.

Example:

```
public class Main{
        public static void main(String args[]){
                int a = 10;
                int b = 20;
                if(a>b){
                        System.out.println(?a is greater than b?);
            }else{
                System.out.println(?b is greater than a?);
            }
             }
}
```

Output:

b is greater than a

If-else-if-ladder:

Syntax:

```
if(Boolean condition){
        // statement1
}else if(Boolean condition){
        // statement2
}else if(Boolean condition){
        // statement3
}
.....
else{
        // else statement
}
```

The if statement executed from top to down. As soon as one of the condition is true, statement associated with that if statement executed.

If none of the condition is true than else statement will be executed, only one of the statement executed from list of else if statements.

Example:

```
public class Main{
        public static void main(String args[]){
                int percentage = 65;
                if(percentage >= 70){
                        System.out.println(?First class with Distinction?);
                }else If(percentage >= 60){
                        System.out.println(?First Class?);
                }else If(percentage >= 48){
                        System.out.println(?Second Class?);
                }else If(percentage >= 36){
                        System.out.println(?Pass Class?);
                }else{
                        System.out.println(?Fail?);
        }

         }
}
```

Output:

First Class

switch statement:

The switch statement is multi way branch statement in java programming. It is use to replace multilevel if-else-if statement.

Syntax:

```
switch(expression){
case value 1:
        // statement 1
        break;
case value 2:
        // statement 2
        break;
case value n:
```

```
        // statement n
        break;
default:
        //statements
        break;
}
```

The expression type must be the byte, short, int and char.

Each case value must be a unique literal(constant not a variable). Duplicate case value not allowed.

The each case value compare with expression if match found than corresponding statement will be executed. If no match is found than default statement will be executed. Default case if optional.

The break statement use to terminate statement sequence, if break statement is not written than all statement execute after match statement.

Example:

```
public class Main{
        public static void main(String args[]){
                int day = 1;
                switch(day){
                        case 0:
                                System.out.println("Sunday");
                                 break;
                        case 1:
                                System.out.println("Monday");
                                break;
                        case 3:
                                System.out.println("Tuesday");
                                break;
                        case 4:
                                System.out.println("Wednesday");
                                break;
                        case 5:
                                System.out.println("Thursday");
                                break;
                        case 6:
                                System.out.println("Friday");
                                break;
                        case 7:
                                System.out.println("Saturday");
                                break;
                        default:
                                System.out.println("Invalid Day");
```

```
                    break;
                }
            }
}
```

Output:
Monday

Note : switch statement support string from Java 7, means use string object in the switch expression.

Numbers Class

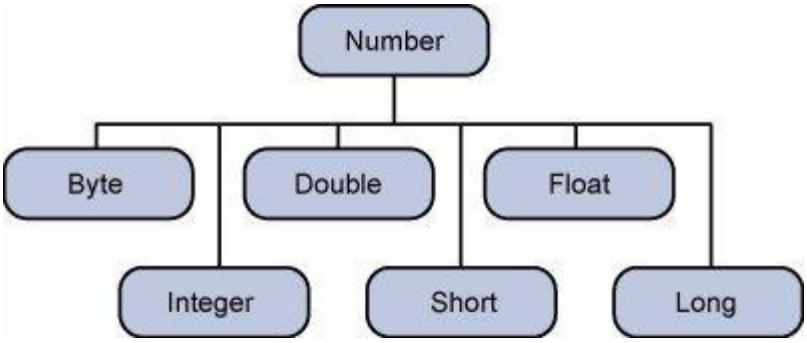
Normally, when we work with Numbers, we use primitive data types such as byte, int, long, double, etc.

Example

```
int i = 5000;

float gpa = 13.65;

double mask = 0xaf;
```

However, in development, we come across situations where we need to use objects instead of primitive data types. In order to achieve this, Java provides **wrapper classes**.

All the wrapper classes (Integer, Long, Byte, Double, Float, Short) are subclasses of the abstract class Number.



Pow():

The method returns the value of the first argument raised to the power of the second argument.

Syntax

```
double pow(double base, double exponent)
```

Parameters

Here is the detail of parameters −

- **base** − Any primitive data type.
- **exponenet** − Any primitive data type.

Return Value

- This method returns the value of the first argument raised to the power of the second argument.

Example

```
public class Test {


  public static void main(String args[]) {
    double x = 11.635;
    double y = 2.76;


    System.out.printf("The value of e is %.4f%n", Math.E);
    System.out.printf("pow(%.3f, %.3f) is %.3f%n", x, y, Math.pow(x, y));
  }
}
```

This will produce the following result −

Output

```
The value of e is 2.7183
pow(11.635, 2.760) is 874.008
```

Sqrt:

The method returns the square root of the argument.

Syntax

```
double sqrt(double d)
```

Parameters

Here is the detail of parameters −

- **d** − Any primitive data type.

Return Value

- This method returns the square root of the argument.

Example

```
public class Test {


  public static void main(String args[]) {
    double x = 11.635;
    double y = 2.76;

    System.out.printf("The value of e is %.4f%n", Math.E);
    System.out.printf("sqrt(%.3f) is %.3f%n", x, Math.sqrt(x));
  }
}
```

This will produce the following result −

Output

```
The value of e is 2.7183
sqrt(11.635) is 3.411
```

Character Methods

Following is the list of the important instance methods that all the subclasses of the Character class implement −

Sr.No.	Method & Description
1	**isLetter()** Determines whether the specified char value is a letter.
2	**isDigit()** Determines whether the specified char value is a digit.
3	**isWhitespace()** Determines whether the specified char value is white space.
4	**isUpperCase()** Determines whether the specified char value is uppercase.
5	**isLowerCase()** Determines whether the specified char value is lowercase.
6	**toUpperCase()** Returns the uppercase form of the specified char value.
7	**toLowerCase()** Returns the lowercase form of the specified char value.
8	**toString()** Returns a String object representing the specified character value that is, a one-character string.

For a complete list of methods, please refer to the java.lang.Character API specification.

String

A string in literal terms is a sequence of characters like the word “hello”. Hey, did you say characters, isn’t it a primitive data type in Java. Yes, so in technical terms, the basic Java String is basically an array of characters. In the java programming language, string is object.

Immutable String

Java String is a immutable object. For an immutable object you cannot modify any of its attribute’s values. Once you have created a java String object it cannot be modified to some other object or a different String. A reference to a java String instance is mutable. There are multiple ways to make an object immutable. Simple and straight forward way is to make all the attributes of that class as final. Java String has all attributes marked as final except hash field.

We all know java String is immutable but do we know why java String is immutable? Main reason behind it is for better performance. Creating a copy of existing java String is easier as there is no need to create a new instance but can be easily created by pointing to already existing String. This saves valuable primary memory.

String Syntax:

String as an array of characters like:

```
char[] charArray = {'V','I','S','I','O','N'};
String str = new String(charArray);
```

String in Java as:

```
String srt = new String("VISION");
```

Initialization of String:

JVM maintains a memory pool for String. When you create a String, first this memory pool is scanned. If the instance already exists then this new instance is mapped to the already existing instance. If not, a new java String instance is created in the memory pool.

```
String str1 = "VISION";
String srt2 = new String();
String str3 = new String("VISION");
String str4 = new String(char[]);
String str5 = new String(byte[]);
String str6 = new String(new StringBuffer());
String str7 = new String(new StringBuilder());
```

We have an empty constructor for String. It is odd, java String is immutable and you have an empty constructur which does nothing but create a empty String. I don't see any use for this constructor, because after you create a String you cannot modify it.

Note: The String class is immutable; so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters, then you should use String Buffer & String Builder Classes.

Example:

```
public class Main{
        public static void main(String args[]){
                String str = new String("visions");
                System.out.println(str);
        }
}
```

Output:
visions

String length:
String length() method returns number of characters contained in the String object.

Example:

```
public class Main{
        public static void main(String args[]){
                String str = new String("visions");
                System.out.println("String length : "+str.length());
        }
}
```

Output:
String length : 7

Concatenating String:
Concatenating is joining of two or more string into one string.
We have two string str1="visions" and str2="developer". If we add these two strings, we should be having a result as str3 ="visionsdeveloper". String having two methods for concatenating string first is "+"(plus operators) and concat() method.

Example:

```
public class Main{
        public static void main(String args[]){
```

```
                String str1 = "visions";
                String str2 = "developer";
                String srt3 = str1+str2;
                System.out.println(str3);
                String str4 = str1.concat(str2);
                System.out.println(str4);
    }
}
```

Output:

visionsdeveloper
visionsdeveloper

Note: Don't use "+" operator because it is not good for performance.

String methods:

String charAt() Method

```
public class Test {


  public static void main(String args[]) {

    String s = "Strings are immutable";

    char result = s.charAt(8);

    System.out.println(result);

  }

}
```

This will produce the following result −

Output

```
A
```

compareToIgnoreCase() Method

```
public class Test {


  public static void main(String args[]) {
```

```
    String str1 = "Strings are immutable";
    String str2 = "Strings are immutable";
    String str3 = "Integers are not immutable";

    int result = str1.compareToIgnoreCase( str2 );
    System.out.println(result);

    result = str2.compareToIgnoreCase( str3 );
    System.out.println(result);

    result = str3.compareToIgnoreCase( str1 );
    System.out.println(result);
  }
}
```

This will produce the following result −

Output

```
0
10
-10
```

String concat() Method

```
public class Test {

  public static void main(String args[]) {
    String s = "Strings are immutable";
    s = s.concat(" all the time");
    System.out.println(s);
  }
```

```
}
```

This will produce the following result −

Output

```
Strings are immutable all the time
```

String copyValueOf() Method

```
public class Test {


  public static void main(String args[]) {
    char[] Str1 = {'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd'};
    String Str2 = "";
    Str2 = Str2.copyValueOf( Str1 );
    System.out.println("Returned String: " + Str2);
  }
}
```

This will produce the following result −

Output

```
Returned String: hello world
```

String indexOf() Method

```
import java.io.*;
public class Test {


  public static void main(String args[]) {
    String Str = new String("Welcome to Tutorialspoint.com");
    System.out.print("Found Index :" );
    System.out.println(Str.indexOf( 'o' ));
  }
```

```
}
```

This will produce the following result −

Output

```
Found Index :4
```

String length() Method

```
import java.io.*;

public class Test {


  public static void main(String args[]) {

    String Str1 = new String("Welcome to Tutorialspoint.com");

    String Str2 = new String("Tutorials" );


    System.out.print("String Length :" );

    System.out.println(Str1.length());


    System.out.print("String Length :" );

    System.out.println(Str2.length());

  }

}
```

This will produce the following result −

Output

```
String Length :29
String Length :9
```

String replace() Method

```
import java.io.*;

public class Test {
```

```
  public static void main(String args[]) {
    String Str = new String("Welcome to Tutorialspoint.com");

    System.out.print("Return Value :" );
    System.out.println(Str.replace('o', 'T'));

    System.out.print("Return Value :" );
    System.out.println(Str.replace('l', 'D'));
  }
}
```

This will produce the following result −

Output

```
Return Value :WelcTme tT TutTrialspTint.cTm
Return Value :WeDcome to TutoriaDspoint.com
```

String split() Method

```
import java.io.*;

public class Test {

   public static void main(String args[]) {
      String Str = new String("Welcome-to-Udamy.com");
      System.out.println("Return Value :" );

      for (String retval: Str.split("-")) {
         System.out.println(retval);
      }
   }
```

```
    }
```

Result:::

```
Return Value :
Welcome
to
Udamy.com
```

String toLowerCase() & toUpperCase()Method

```
import java.io.*;

public class Test {


  public static void main(String args[]) {

    String Str = new String("Welcome to Pau");


    System.out.print("Return Value :");

    System.out.println(Str.toLowerCase());

    System.out.println(Str.toUpperCase() );

  }

}
```

Result:-

```
Return Value :welcome to pau
WELCOME TO PAU
```

Arrays

Declaring Array Variables

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable −

Syntax

```
dataType[] arrayRefVar;   // preferred way.
or
dataType arrayRefVar[];  // works but not preferred way.
```

Note – The style **dataType[] arrayRefVar** is preferred. The style **dataType arrayRefVar[]** comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers.

Example

The following code snippets are examples of this syntax –

```
double[] myList;   // preferred way.

or

double myList[];   // works but not preferred way.
```

Creating Arrays

You can create an array by using the new operator with the following syntax –

Syntax

```
arrayRefVar = new dataType[arraySize];
```

The above statement does two things –

- It creates an array using new dataType[arraySize].
- It assigns the reference of the newly created array to the variable arrayRefVar.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below –

```
dataType[] arrayRefVar = new dataType[arraySize];
```

Alternatively you can create arrays as follows –

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

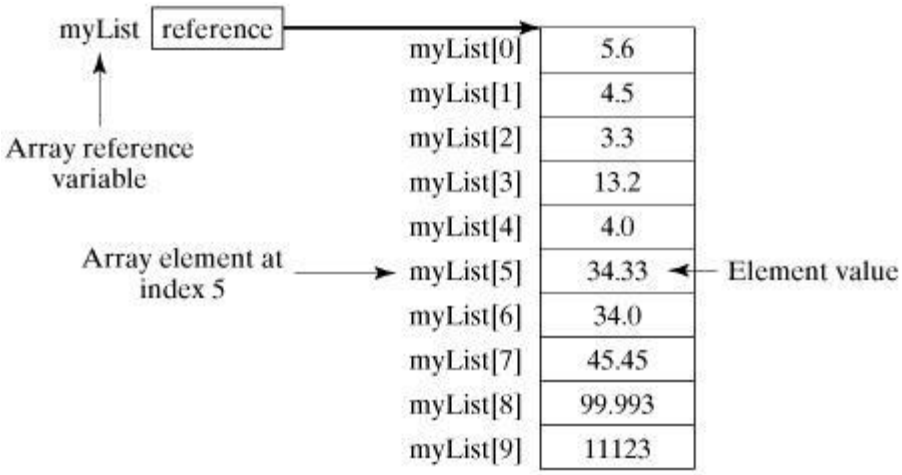
The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

Example

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList –

```
double[] myList = new double[10];
```

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.



Processing Arrays

When processing array elements, we often use either **for** loop or **foreach** loop because all of the elements in an array are of the same type and the size of the array is known.

Example

Here is a complete example showing how to create, initialize, and process arrays –

```
public class TestArray {


  public static void main(String[] args) {
    double[] myList = {1.9, 2.9, 3.4, 3.5};


    // Print all the array elements
    for (int i = 0; i < myList.length; i++) {
      System.out.println(myList[i] + " ");
    }


    // Summing all elements
    double total = 0;
    for (int i = 0; i < myList.length; i++) {
      total += myList[i];
    }
```

```
    System.out.println("Total is " + total);


    // Finding the largest element
    double max = myList[0];
    for (int i = 1; i < myList.length; i++) {
      if (myList[i] > max) max = myList[i];
    }
    System.out.println("Max is " + max);
  }
}
```

This will produce the following result −

Output

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

The foreach Loops

JDK 1.5 introduced a new for loop known as foreach loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable.

Example

The following code displays all the elements in the array myList −

```
public class TestArray {


  public static void main(String[] args) {
    double[] myList = {1.9, 2.9, 3.4, 3.5};


    // Print all the array elements
    for (double element: myList) {
```

```
      System.out.println(element);
    }
  }
}
```

This will produce the following result −

Output

```
1.9
2.9
3.4
3.5
```

Passing Arrays to Methods

Just as you can pass primitive type values to methods, you can also pass arrays to methods. For example, the following method displays the elements in an **int** array −

Example

```
public static void printArray(int[] array) {
  for (int i = 0; i < array.length; i++) {
    System.out.print(array[i] + " ");
  }
}
```

You can invoke it by passing an array. For example, the following statement invokes the printArray method to display 3, 1, 2, 6, 4, and 2 −

Example

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

Returning an Array from a Method

A method may also return an array. For example, the following method returns an array that is the reversal of another array −

Example

```
public static int[] reverse(int[] list) {
  int[] result = new int[list.length];
```

```
  for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {
    result[j] = list[i];
  }
  return result;
}
```

An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed.

The first element of array start with zero.

Using an array in your program is a 3 step process:

1. Declaring you array.
2. Constructing your array.
3. Initializing your array.

Declaring Array:-

Syntax:

```
elementType[] arrayName;
Or
elementType arrayName[];
```

Example:

```
int[] intArray;
int intArray[];
```

Constructing Array:-

Syntax:

```
new  elementType[size];
```

Example:

```
int[] intArray = new int[10]; // Defines that intArray will store 10 integer values
int intArray[] = new int[10];
```

Initializing Array:-

Syntax:

```
arrayName[element 0,1,2?.. N] = value;
```

Example:

```
intArray[0] = 10; // Assign an integer value 10 to the first element 0 of the array
intArray[1] = 20;
```

Declaring and Initializing Array:-

Syntax:

```
elementType[] arrayName = {values1,values2,? valueN};
```

Example:

```
int[] intArray = {1,2,3,4};
```

Array Example:

```
public class Main {
 public static void main(String[] args) {

  String[] names = new String[3];
  names[0] = "A";
  names[1] = "ABC";
  names[2] = "XYZ";
  for (int i = 0; i < names.length; i++) {
   System.out.println(names[i]);
  }
  //this line should throw an exception
  //System.out.println(names[6]);
 }
}
```

Array are passed by refrence:

Arrays are passed to functions by reference, or as a pointer to the original. This means anything you do to the Array inside the function affects the original.

Example:

```
public class Main{
        public static void passByRefrence(String a[]){
                a[0] = "Z";
```

```
        }
        public static void main(String args[]){
                String[] b = {"A","B","C"};
                System.out.println("Before Function call  :  "+b[0]);
                Main.passByRefrence(b);
                System.out.println("After Function call : "+b[0]);
        }
}
```

Output:

Before Function call : A

After Function call : Z

Multidimensional Arrays:

Multidimensional arrays, are arrays of arrays.

Syntax:

```
elementType[][] arrayName = new elementType[size][size];
```

Example:

```
int[][] intArrays = new int[4][5];
```

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension.

You can allocate the remaining dimensions separately.

In Java the length of each array in a multidimensional array is under your control.

Example:

```
int multi[][] = new int[2][];
multi[0] = new int[5];
multi[1] = new int[4];
```

Array of Objects:

It is possible to create array of objects of user created class.

Example:

```
class Employee{
        int id;
        String name;
        public void setData(int id, String name){
                this.id = id;
                this.name = name;
        }
```

```
        public void displayData(){
                System.out.println("Employee ID : "+this.id);
                System.out.println("Employee Name : "+this.name);
        }
}
class Main{
        public static void main(String args[]){
                Employee[] emp = new Employee[2];
                emp[0].setData(1,"ABC");
                emp[1].setData(2,"XYZ");
                emp[0].displayData();
                emp[1].displayData();
        }
}
```

Output:
Employee ID : 1
Employee Name : ABC
Employee ID : 2
Employee Name : XYZ

10 methods of array.

0. Declare an array

```
String[] aArray = new String[5];
String[] bArray = {"a","b","c", "d", "e"};
String[] cArray = new String[]{"a","b","c","d","e"};
```

1. Print an array in Java

```
int[] intArray = { 1, 2, 3, 4, 5 };
String intArrayString = Arrays.toString(intArray);
// print directly will print reference value
System.out.println(intArray);
// [I@7150bd4d
System.out.println(intArrayString);
// [1, 2, 3, 4, 5]
```

2. Create an ArrayList from an array

```
String[] stringArray = { "a", "b", "c", "d", "e" };
ArrayList<String> arrayList = new ArrayList<String>(Arrays.asList(stringArray));
System.out.println(arrayList);
// [a, b, c, d, e]
```

3. Check if an array contains a certain value

```
String[] stringArray = { "a", "b", "c", "d", "e" };
boolean b = Arrays.asList(stringArray).contains("a");
System.out.println(b);
// true
```

4. Concatenate two arrays

```
int[] intArray = { 1, 2, 3, 4, 5 };
int[] intArray2 = { 6, 7, 8, 9, 10 };
// Apache Commons Lang library
int[] combinedIntArray = ArrayUtils.addAll(intArray, intArray2);
```

5. Declare an array inline

```
method(new String[]{"a", "b", "c", "d", "e"});
```

6. Joins the elements of the provided array into a single String

```
// containing the provided list of elements
// Apache common lang
String j = StringUtils.join(new String[] { "a", "b", "c" }, ", ");
System.out.println(j);
// a, b, c
```

7. Covnert an ArrayList to an array

```
String[] stringArray = { "a", "b", "c", "d", "e" };
ArrayList<String> arrayList = new ArrayList<String>(Arrays.asList(stringArray));
String[] stringArr = new String[arrayList.size()];
arrayList.toArray(stringArr);
for (String s : stringArr)
        System.out.println(s);
```

8. Convert an array to a set

```
Set<String> set = new HashSet<String>(Arrays.asList(stringArray));
System.out.println(set);
//[d, e, b, c, a]
```

9. Reverse an array

```
int[] intArray = { 1, 2, 3, 4, 5 };
ArrayUtils.reverse(intArray);
System.out.println(Arrays.toString(intArray));
//[5, 4, 3, 2, 1]
```

10. Remove element of an array

```
int[] intArray = { 1, 2, 3, 4, 5 };
int[] removed = ArrayUtils.removeElement(intArray, 3);//create a new array
System.out.println(Arrays.toString(removed));
```

One more - convert int to byte array

```
byte[] bytes = ByteBuffer.allocate(4).putInt(8).array();

for (byte t : bytes) {
  System.out.format("0x%x ", t);
}
```


ArrayList

ArrayList in Java is used to store dynamically sized collection of elements. Contrary to Arrays that are fixed in size, an ArrayList grows its size automatically when new elements are added to it.
ArrayList is part of Java's collection framework and implements Java's `List` interface.

Following are few key points to note about ArrayList in Java -

- An ArrayList is a re-sizable array, also called a dynamic array. It grows its size to accommodate new elements and shrinks the size when the elements are removed.
- ArrayList internally uses an array to store the elements. Just like arrays, It allows you to retrieve the elements by their index.
- Java ArrayList allows duplicate and null values.
- Java ArrayList is an ordered collection. It maintains the insertion order of the elements.
- You cannot create an ArrayList of primitive types like `int`, `char` etc. You need to use boxed types like `Integer`, `Character`, `Boolean` etc.
- Java ArrayList is not synchronized. If multiple threads try to modify an ArrayList at the same time, then the final outcome will be non-deterministic. You must explicitly synchronize access to an ArrayList if multiple threads are gonna modify it.

Creating an ArrayList and adding new elements to it
This example shows:

- How to create an ArrayList using the `ArrayList()` constructor.
- Add new elements to an ArrayList using the `add()` method.

```
import java.util.ArrayList;

import java.util.List;

public class CreateArrayListExample {

   public static void main(String[] args) {

      // Creating an ArrayList of String

      List<String> animals = new ArrayList<>();

      // Adding new elements to the ArrayList

      animals.add("Lion");

      animals.add("Tiger");

      animals.add("Cat");

      animals.add("Dog");

      System.out.println(animals);
```

```
        // Adding an element at a particular index in an ArrayList
        animals.add(2, "Elephant");
        System.out.println(animals);
    }
}
# Output
[Lion, Tiger, Cat, Dog]
[Lion, Tiger, Elephant, Cat, Dog]
```

Creating an ArrayList from another collection

This example shows:

- How to create an ArrayList from another collection using the `ArrayList(Collection c)` constructor.
- How to add all the elements from an existing collection to the new ArrayList using the `addAll()` method.import java.util.ArrayList;

```
import java.util.List;
public class CreateArrayListFromCollectionExample {
    public static void main(String[] args) {
        List<Integer> firstFivePrimeNumbers = new ArrayList<>();
        firstFivePrimeNumbers.add(2);
        firstFivePrimeNumbers.add(3);
        firstFivePrimeNumbers.add(5);
        firstFivePrimeNumbers.add(7);
        firstFivePrimeNumbers.add(11);
        // Creating an ArrayList from another collection
        List<Integer> firstTenPrimeNumbers = new ArrayList<>(firstFivePrimeNumbers);
        List<Integer> nextFivePrimeNumbers = new ArrayList<>();
        nextFivePrimeNumbers.add(13);
        nextFivePrimeNumbers.add(17);
```

```
        nextFivePrimeNumbers.add(19);
        nextFivePrimeNumbers.add(23);
        nextFivePrimeNumbers.add(29);
        // Adding an entire collection to an ArrayList
        firstTenPrimeNumbers.addAll(nextFivePrimeNumbers);
        System.out.println(firstTenPrimeNumbers);
    }
}
# Output
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Accessing elements from an ArrayList
This example shows:

- How to check if an ArrayList is empty using the `isEmpty()` method.
- How to find the size of an ArrayList using the `size()` method.
- How to access the element at a particular index in an ArrayList using the `get()` method.
- How to modify the element at a particular index in an ArrayList using the `set()` method.

```
import java.util.ArrayList;
import java.util.List;
public class AccessElementsFromArrayListExample {
    public static void main(String[] args) {
        List<String> topCompanies = new ArrayList<>();
        // Check if an ArrayList is empty
        System.out.println("Is the topCompanies list empty? : " + topCompanies.isEmpty());
        topCompanies.add("Google");
        topCompanies.add("Apple");
        topCompanies.add("Microsoft");
        topCompanies.add("Amazon");
        topCompanies.add("Facebook");
        // Find the size of an ArrayList
        System.out.println("Here are the top " + topCompanies.size() + " companies in the
world");
        System.out.println(topCompanies);
```

```
        // Retrieve the element at a given index
        String bestCompany = topCompanies.get(0);
        String secondBestCompany = topCompanies.get(1);
        String lastCompany = topCompanies.get(topCompanies.size() - 1);
        System.out.println("Best Company: " + bestCompany);
        System.out.println("Second Best Company: " + secondBestCompany);
        System.out.println("Last Company in the list: " + lastCompany);
        // Modify the element at a given index
        topCompanies.set(4, "Walmart");
        System.out.println("Modified top companies list: " + topCompanies);
    }
}
# Output

Is the topCompanies list empty? : true

Here are the top 5 companies in the world

[Google, Apple, Microsoft, Amazon, Facebook]

Best Company: Google

Second Best Company: Apple
Last Company in the list: Facebook
Modified top companies list: [Google, Apple, Microsoft, Amazon, Walmart]
```

Removing elements from an ArrayList

This example shows:

1. How to remove the element at a given index in an ArrayList | remove(int index)
2. How to remove an element from an ArrayList | remove(Object o)
3. How to remove all the elements from an ArrayList that exist in a given collection | removeAll()
4. How to remove all the elements matching a given predicate | removeIf()
5. How to clear an ArrayList | clear()


```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;
public class RemoveElementsFromArrayListExample {
    public static void main(String[] args) {
        List<String> programmingLanguages = new ArrayList<>();
        programmingLanguages.add("C");
        programmingLanguages.add("C++");
```

```
        programmingLanguages.add("Java");
        programmingLanguages.add("Kotlin");
        programmingLanguages.add("Python");
        programmingLanguages.add("Perl");
        programmingLanguages.add("Ruby");
        System.out.println("Initial List: " + programmingLanguages);
        // Remove the element at index `5`
        programmingLanguages.remove(5);
        System.out.println("After remove(5): " + programmingLanguages);
        // Remove the first occurrence of the given element from the ArrayList
        // (The remove() method returns false if the element does not exist in the ArrayList)
        boolean isRemoved = programmingLanguages.remove("Kotlin");
        System.out.println("After remove(\"Kotlin\"): " + programmingLanguages);
        // Remove all the elements that exist in a given collection
        List<String> scriptingLanguages = new ArrayList<>();
        scriptingLanguages.add("Python");
        scriptingLanguages.add("Ruby");
        scriptingLanguages.add("Perl");
        programmingLanguages.removeAll(scriptingLanguages);
        System.out.println("After removeAll(scriptingLanguages): " + programmingLanguages);
        // Remove all the elements that satisfy the given predicate
        programmingLanguages.removeIf(new Predicate<String>() {
            @Override
            public boolean test(String s) {
                return s.startsWith("C");
            }
        System.out.println("After Removing all elements that start with \"C\": " +
programmingLanguages);
        // Remove all elements from the ArrayList
        programmingLanguages.clear();
        System.out.println("After clear(): " + programmingLanguages);
    }
}
# Output
```

```
Initial List: [C, C++, Java, Kotlin, Python, Perl, Ruby]
After remove(5): [C, C++, Java, Kotlin, Python, Ruby]
After remove("Kotlin"): [C, C++, Java, Python, Ruby]
After removeAll(scriptingLanguages): [C, C++, Java]
After Removing all elements that start with "C": [Java]
After clear(): []
```

Iterating over an ArrayList

The following example shows how to iterate over an ArrayList using

1. Java 8 `forEach` and lambda expression.
2. `iterator()`.
3. `iterator()` and Java 8 `forEachRemaining()` method.
4. `listIterator()`.
5. Simple for-each loop.
6. for loop with index.

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.ListIterator;
public class IterateOverArrayListExample {
   public static void main(String[] args) {
      List<String> tvShows = new ArrayList<>();
      tvShows.add("Breaking Bad");
      tvShows.add("Game Of Thrones");
      tvShows.add("Friends");
      tvShows.add("Prison break");
      System.out.println("=== Iterate using Java 8 forEach and lambda ===");
      tvShows.forEach(tvShow -> {
         System.out.println(tvShow);
```

```
        });
        System.out.println("\n=== Iterate using an iterator() ===");
        Iterator<String> tvShowIterator = tvShows.iterator();
        while (tvShowIterator.hasNext()) {
            String tvShow = tvShowIterator.next();
            System.out.println(tvShow);
        }
        System.out.println("\n=== Iterate using an iterator() and Java 8 forEachRemaining()
method ===");
        tvShowIterator = tvShows.iterator();
        tvShowIterator.forEachRemaining(tvShow -> {
            System.out.println(tvShow);
        });
        System.out.println("\n=== Iterate using a listIterator() to traverse in both directions
===");
        // Here, we start from the end of the list and traverse backwards.
        ListIterator<String> tvShowListIterator = tvShows.listIterator(tvShows.size());
        while (tvShowListIterator.hasPrevious()) {
            String tvShow = tvShowListIterator.previous();
            System.out.println(tvShow);
        }
        System.out.println("\n=== Iterate using simple for-each loop ===");
        for(String tvShow: tvShows) {
            System.out.println(tvShow);
        }

        System.out.println("\n=== Iterate using for loop with index ===");
        for(int i = 0; i < tvShows.size(); i++) {
```

```
            System.out.println(tvShows.get(i));
        }
    }
}
```

```
# Output
=== Iterate using Java 8 forEach and lambda ===
Breaking Bad
Game Of Thrones
Friends
Prison break
=== Iterate using an iterator() ===
Breaking Bad
Game Of Thrones
Friends
Prison break
=== Iterate using an iterator() and Java 8 forEachRemaining() method ===
Breaking Bad
Game Of Thrones
Friends
Prison break
=== Iterate using a listIterator() to traverse in both directions ===
Prison break
Friends
Game Of Thrones
Breaking Bad
=== Iterate using simple for-each loop ===
Breaking Bad
Game Of Thrones
Friends
Prison break
=== Iterate using for loop with index ===
Breaking Bad
```

```
Game Of Thrones
Friends
Prison break
```

The `iterator()` and `listIterator()` methods are useful when you need to modify the ArrayList while traversing.

Consider the following example, where we remove elements from the ArrayList using `iterator.remove()` method while traversing through it -


```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class ArrayListIteratorRemoveExample {
   public static void main(String[] args) {
      List<Integer> numbers = new ArrayList<>();
      numbers.add(13);
      numbers.add(18);
      numbers.add(25);
      numbers.add(40);

      Iterator<Integer> numbersIterator = numbers.iterator();
      while (numbersIterator.hasNext()) {
         Integer num = numbersIterator.next();
         if(num % 2 != 0) {
            numbersIterator.remove();
         }
      }
      System.out.println(numbers);
   }
}
# Output
[18, 40]
```

Searching for elements in an ArrayList

The example below shows how to:

- Check if an ArrayList contains a given element | contains()
- Find the index of the first occurrence of an element in an ArrayList | indexOf()
- Find the index of the last occurrence of an element in an ArrayList | lastIndexOf()

```
import java.util.ArrayList;
import java.util.List;
public class SearchElementsInArrayListExample {
   public static void main(String[] args) {
      List<String> names = new ArrayList<>();
      names.add("John");
      names.add("Alice");
      names.add("Bob");
      names.add("Steve");
      names.add("John");
      names.add("Steve");
      names.add("Maria");
      // Check if an ArrayList contains a given element
      System.out.println("Does names array contain \"Bob\"? : " + names.contains("Bob"));
      // Find the index of the first occurrence of an element in an ArrayList
      System.out.println("indexOf \"Steve\": " + names.indexOf("Steve"));
      System.out.println("indexOf \"Mark\": " + names.indexOf("Mark"));
      // Find the index of the last occurrence of an element in an ArrayList
      System.out.println("lastIndexOf \"John\" : " + names.lastIndexOf("John"));
      System.out.println("lastIndexOf \"Bill\" : " + names.lastIndexOf("Bill"));
   }
}
# Output
Does names array contain "Bob"? : true
indexOf "Steve": 3
indexOf "Mark": -1
lastIndexOf "John" : 4
lastIndexOf "Bill" : -1
```

ArrayList of user defined objects

Since ArrayList supports generics, you can create an ArrayList of **any** type. It can be of simple types like `Integer`, `String`, `Double` or complex types like an ArrayList of ArrayLists, or an ArrayList of HashMaps or an ArrayList of any user defined objects.

In the following example, you'll learn how to create an ArrayList of user defined objects.


```
import java.util.ArrayList;
import java.util.List;
class User {
   private String name;
   private int age;
   public User(String name, int age) {
      this.name = name;
      this.age = age;
   }

   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public int getAge() {
      return age;
   }
   public void setAge(int age) {
      this.age = age;
   }
}
public class ArrayListUserDefinedObjectExample {
   public static void main(String[] args) {
      List<User> users = new ArrayList<>();
      users.add(new User("Rajeev", 25));
      users.add(new User("John", 34));
      users.add(new User("Steve", 29));
```

```
        users.forEach(user -> {
            System.out.println("Name : " + user.getName() + ", Age : " + user.getAge());
        });
    }
}
# Output
Name : Rajeev, Age : 25
Name : John, Age : 34
Name : Steve, Age : 29
```

Sorting an ArrayList

Sorting an ArrayList is a very common task that you will encounter in your programs. In this section, I'll show you how to -

- Sort an ArrayList using `Collections.sort()` method.
- Sort an ArrayList using `ArrayList.sort()` method.
- Sort an ArrayList of user defined objects with a custom comparator.

1. Sort an ArrayList using Collections.sort() method

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class ArrayListCollectionsSortExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>();
        numbers.add(13);
        numbers.add(7);
        numbers.add(18);
        numbers.add(5);
        numbers.add(2);
        System.out.println("Before : " + numbers);
        // Sorting an ArrayList using Collections.sort() method
        Collections.sort(numbers);
        System.out.println("After : " + numbers);
    }
}
# Output
Before : [13, 7, 18, 5, 2]
After : [2, 5, 7, 13, 18]
```

2. Sort an ArrayList using ArrayList.sort() method

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
public class ArrayListSortExample {
   public static void main(String[] args) {
      List<String> names = new ArrayList<>();
      names.add("Lisa");
      names.add("Jennifer");
      names.add("Mark");
      names.add("David");
      System.out.println("Names : " + names);
      // Sort an ArrayList using its sort() method. You must pass a Comparator to the
ArrayList.sort() method.
      names.sort(new Comparator<String>() {
         @Override
         public int compare(String name1, String name2) {
            return name1.compareTo(name2);
         }
      });
      // The above `sort()` method call can also be written simply using lambda expression
      names.sort((name1, name2) -> name1.compareTo(name2));
      // Following is an even more concise solution
      names.sort(Comparator.naturalOrder());
      System.out.println("Sorted Names : " + names);
   }
}
# Output
Names : [Lisa, Jennifer, Mark, David]
Sorted Names : [David, Jennifer, Lisa, Mark]
```


3. Sort an ArrayList of Objects using custom Comparator

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
class Person {
   private String name;
   private Integer age;
   public Person(String name, Integer age) {
      this.name = name;
      this.age = age;
   }
```

```
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Integer getAge() {
        return age;
    }
    public void setAge(Integer age) {
        this.age = age;
    }
    @Override
    public String toString() {
        return "{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
public class ArrayListObjectSortExample {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Sachin", 47));
        people.add(new Person("Chris", 34));
        people.add(new Person("Rajeev", 25));
        people.add(new Person("David", 31));
        System.out.println("Person List : " + people);
        // Sort People by their Age
        people.sort((person1, person2) -> {
            return person1.getAge() - person2.getAge();
        });
        // A more concise way of writing the above sorting function
        people.sort(Comparator.comparingInt(Person::getAge));
        System.out.println("Sorted Person List by Age : " + people);
        // You can also sort using Collections.sort() method by passing the custom Comparator
        Collections.sort(people, Comparator.comparing(Person::getName));
        System.out.println("Sorted Person List by Name : " + people);
    }
}
# Output
Person List : [{name='Sachin', age=47}, {name='Chris', age=34}, {name='Rajeev', age=25},
{name='David', age=31}]
Sorted Person List by Age : [{name='Rajeev', age=25}, {name='David', age=31},
{name='Chris', age=34}, {name='Sachin', age=47}]
```

```
Sorted Person List by Name : [{name='Chris', age=34}, {name='David', age=31},
{name='Rajeev', age=25}, {name='Sachin', age=47}]
```

Synchronizing Access to an ArrayList

The ArrayList class is not synchronized. If multiple threads try to modify an ArrayList at the same time then the final result becomes not-deterministic because one thread might override the changes done by another thread.

Example Demonstrating ArrayList's unpredictable behavior in multi-threaded environments

The following example shows what happens when multiple threads try to modify an ArrayList at the same time.


```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
public class UnsafeArrayListExample {
   public static void main(String[] args) throws InterruptedException {
      List<Integer> unsafeArrayList = new ArrayList<>();
      unsafeArrayList.add(1);
      unsafeArrayList.add(2);
      unsafeArrayList.add(3);
      // Create a thread pool of size 10
      ExecutorService executorService = Executors.newFixedThreadPool(10);
      // Create a Runnable task that increments each element of the ArrayList by one
      Runnable task = () -> {
         incrementArrayList(unsafeArrayList);
      };
      // Submit the task to the executor service 100 times.
      // All the tasks will modify the ArrayList concurrently
      for(int i = 0; i < 100; i++) {
         executorService.submit(task);
```

```
        }
        executorService.shutdown();
        executorService.awaitTermination(60, TimeUnit.SECONDS);
        System.out.println(unsafeArrayList);
    }
    // Increment all the values in the ArrayList by one
    private static void incrementArrayList(List<Integer> unsafeArrayList) {
        for(int i = 0; i < unsafeArrayList.size(); i++) {
            Integer value = unsafeArrayList.get(i);
            unsafeArrayList.set(i, value + 1);
        }
    }
}
```

96

The final output of the above program should be equal to `[101, 102, 103]` because we're incrementing the values in the ArrayList 100 times. But if you run the program, it will produce different output every time it is run -

```
# Output
[96, 96, 98]
```

Try running the above program multiple times and see how it produces different outputs. To learn more about such issues in multi-threaded programs, check out my article on Java Concurrency Issues and Thread Synchronization.

Example demonstrating how to synchronize concurrent modifications to an ArrayList

All right! Now let's see how we can synchronize access to the `ArrayList` in multi-threaded environments.

The following example shows the synchronized version of the previous example. Unlike the previous program, the output of this program is deterministic and will always be the same.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.ExecutorService;
```

```
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
public class SynchronizedArrayListExample {
    public static void main(String[] args) throws InterruptedException {
        List<Integer> safeArrayList = Collections.synchronizedList(new ArrayList<>());
        safeArrayList.add(1);
        safeArrayList.add(2);
        safeArrayList.add(3);
        // Create a thread pool of size 10
        ExecutorService executorService = Executors.newFixedThreadPool(10);
        // Create a Runnable task that increments each element of the ArrayList by one
        Runnable task = () -> {
            incrementArrayList(safeArrayList);
        };
        // Submit the task to the executor service 100 times.
        // All the tasks will modify the ArrayList concurrently
        for(int i = 0; i < 100; i++) {
            executorService.submit(task);
        }

        executorService.shutdown();
        executorService.awaitTermination(60, TimeUnit.SECONDS);
        System.out.println(safeArrayList);
    }
    // Increment all the values in the ArrayList by one
    private static void incrementArrayList(List<Integer> safeArrayList) {
        synchronized (safeArrayList) {
            for (int i = 0; i < safeArrayList.size(); i++) {
                Integer value = safeArrayList.get(i);
                safeArrayList.set(i, value + 1);
            }
        }
    }
}
```

Output

[101, 102, 103]

Vector in Java

Vector implements List Interface. Like ArrayList it also maintains insertion order but it is rarely used in non-thread environment as it is synchronized and due to which it gives poor performance in searching, adding, delete and update of its elements.

Three ways to create vector class object:

Method 1:

```
Vector vec = new Vector();
```

It creates an empty Vector with the default initial capacity of 10. It means the Vector will be re-sized when the 11th elements needs to be inserted into the Vector. Note: By default vector doubles its size. i.e. In this case the Vector size would remain 10 till 10 insertions and once we try to insert the 11th element It would become 20 (double of default capacity 10).

Method 2:

Syntax: Vector object= new Vector(int initialCapacity)

```
Vector vec = new Vector(3);
```

It will create a Vector of initial capacity of 3.

Method 3:

Syntax:

```
Vector object= new vector(int initialcapacity, capacityIncrement)
```

Example:

```
Vector vec= new Vector(4, 6)
```

Here we have provided two arguments. The initial capacity is 4 and capacityIncrement is 6. It means upon insertion of 5th element the size would be 10 (4+6) and on 11th insertion it would be 16(10+6).

Complete Example of Vector in Java:

```
import java.util.*;
public class VectorExample {
  public static void main(String args[]) {
    /* Vector of initial capacity(size) of 2 */
    Vector<String> vec = new Vector<String>(2);
    /* Adding elements to a vector*/
    vec.addElement("Apple");
    vec.addElement("Orange");
    vec.addElement("Mango");
    vec.addElement("Fig");
    /* check size and capacityIncrement*/
    System.out.println("Size is: "+vec.size());
```

```
    System.out.println("Default capacity increment is: "+vec.capacity());
    vec.addElement("fruit1");
    vec.addElement("fruit2");
    vec.addElement("fruit3");
    /*size and capacityIncrement after two insertions*/
    System.out.println("Size after addition: "+vec.size());
    System.out.println("Capacity after increment is: "+vec.capacity());
    /*Display Vector elements*/
    Enumeration en = vec.elements();
    System.out.println("\nElements are:");
    while(en.hasMoreElements())
      System.out.print(en.nextElement() + " ");
  }
}
```


Output:

```
Size is: 4
Default capacity increment is: 4
Size after addition: 7
Capacity after increment is: 8

Elements are:
Apple Orange Mango Fig fruit1 fruit2 fruit3
```

Commonly used methods of Vector Class:

1. **void addElement(Object element):** It inserts the element at the end of the Vector.
2. **int capacity():** This method returns the current capacity of the vector.
3. **int size():** It returns the current size of the vector.
4. **void setSize(int size):** It changes the existing size with the specified size.
5. **boolean contains(Object element):** This method checks whether the specified element is present in the Vector. If the element is been found it returns true else false.
6. **boolean containsAll(Collection c):** It returns true if all the elements of collection c are present in the Vector.
7. **Object elementAt(int index):** It returns the element present at the specified location in Vector.
8. **Object firstElement():** It is used for getting the first element of the vector.
9. **Object lastElement():** Returns the last element of the array.
10. **Object get(int index):** Returns the element at the specified index.
11. **boolean isEmpty():** This method returns true if Vector doesn't have any element.
12. **boolean removeElement(Object element):** Removes the specifed element from vector.
13. **boolean removeAll(Collection c):** It Removes all those elements from vector which are present in the Collection c.
14. **void setElementAt(Object element, int index):** It updates the element of specifed index with the given element.

Difference between ArrayList and Vector

ArrayList and Vector both implements List interface and maintains insertion order.

But there are many differences between ArrayList and Vector classes that are given belo

100

ArrayList	Vector
1) ArrayList is **not synchronized**.	Vector is **synchronized**.
2) ArrayList **increments 50%** of current array size if number of element exceeds from its capacity.	Vector **increments 100%** means doubles the array size if total number of element exceeds than its capacity.
3) ArrayList is **not a legacy** class, it is introduced in JDK 1.2.	Vector is a **legacy** class.
4) ArrayList is **fast** because it is non-synchronized.	Vector is **slow** because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object.
5) ArrayList uses **Iterator** interface to traverse the elements.	Vector uses **Enumeration** interface to traverse the elements. But it can use Iterator also.

Example of Java ArrayList

Let's see a simple example where we are using ArrayList to store and traverse the elements.

```
1. import java.util.*;
2. class TestArrayList21{
3.  public static void main(String args[]){
4.
5.   List<String> al=new ArrayList<String>();//creating arraylist
6.   al.add("Sonoo");//adding object in arraylist
7.   al.add("Michael");
8.   al.add("James");
9.   al.add("Andy");
```

```
10.  //traversing elements using Iterator
11.  Iterator itr=al.iterator();
12.  while(itr.hasNext()){
13.   System.out.println(itr.next());
14.  }
15. }
16. }
```


Output:

```
Sonoo
Michael
James
Andy
```

Example of Java Vector

Let's see a simple example of java Vector class that uses Enumeration interface.

```
1.  import java.util.*;
2.  class TestVector1{
3.   public static void main(String args[]){
4.    Vector<String> v=new Vector<String>();//creating vector
5.    v.add("umesh");//method of Collection
6.    v.addElement("irfan");//method of Vector
7.    v.addElement("kumar");
8.    //traversing elements using Enumeration
9.    Enumeration e=v.elements();
10.   while(e.hasMoreElements()){
11.    System.out.println(e.nextElement());
12.   }
13.  }
14. }
```

Output:

```
umesh
irfan
kumar
```

Date and Time

Getting Current Date and Time:

This is a very easy method to get current date and time in Java. You can use a simple Date object with *toString()* method to print the current date and time as follows −

Example

```
import java.util.Date;

public class DateDemo {


  public static void main(String args[]) {
    // Instantiate a Date object
    Date date = new Date();


    // display time and date using toString()
    System.out.println(date.toString());
  }
}
```


This will produce the following result −

Output

```
on May 04 12:04:52 CDT 2018
```

Java LocalDate Example

```
1. import java.time.LocalDate;
2. public class LocalDateExample1 {
3.   public static void main(String[] args) {
4.     LocalDate date = LocalDate.now();
5.     LocalDate yesterday = date.minusDays(1);
6.     LocalDate tomorrow = yesterday.plusDays(2);
7.     System.out.println("Today date: "+date);
8.     System.out.println("Yesterday date: "+yesterday);
9.     System.out.println("Tommorow date: "+tomorrow);
10.  }
11. }
```

Output:

```
Today date: 2017-01-13
```

```
Yesterday date: 2017-01-12
Tommorow date: 2017-01-14
```

Java LocalDate Example: isLeapYear()

```
1. import java.time.LocalDate;
2. public class LocalDateExample2 {
3.   public static void main(String[] args) {
4.     LocalDate date1 = LocalDate.of(2017, 1, 13);
5.     System.out.println(date1.isLeapYear());
6.     LocalDate date2 = LocalDate.of(2016, 9, 23);
7.     System.out.println(date2.isLeapYear());
8.   }
9. }
```


Output:

```
false
true
```

Java LocalDate Example: atTime()

```
1. import java.time.*;
2. public class LocalDateExample3 {
3.   public static void main(String[] args) {
4.     LocalDate date = LocalDate.of(2017, 1, 13);
5.     LocalDateTime datetime = date.atTime(1,50,9);
6.     System.out.println(datetime);
7.   }
8. }
```

Output:

```
2017-01-13T01:50:09
```

Get Current Date and Time in Java

There are many ways to get current date and time in java. There are many classes that can be used to get current date and time in java.

1. java.time.format.DateTimeFormatter class
2. java.text.SimpleDateFormat class
3. java.time.LocalDate class

4. java.time.LocalTime class
5. java.time.LocalDateTime class
6. java.time.Clock class
7. java.util.Date class
8. java.sql.Date class
9. java.util.Calendar class

Get Current Date and Time: java.time.format.DateTimeFormatter

The LocalDateTime.now() method returns the instance of LocalDateTime class. If we print the instance of LocalDateTime class, it prints current date and time. To format the current date, you can use DateTimeFormatter class which is included in JDK 1.8.

```
1. import java.time.format.DateTimeFormatter;
2. import java.time.LocalDateTime;
3. public class CurrentDateTimeExample1 {
4.   public static void main(String[] args) {
5.    DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
6.    LocalDateTime now = LocalDateTime.now();
7.    System.out.println(dtf.format(now));
8.   }
9. }
```

Output:

```
2017/11/06 12:11:58
```

Get Current Date and Time: java.text.SimpleDateFormat

The SimpleDateFormat class is also used for formatting date and time. But it is old approach.

```
1. import java.text.SimpleDateFormat;
2. import java.util.Date;
3. public class CurrentDateTimeExample2 {
4. public static void main(String[] args) {
5.    SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
6.    Date date = new Date();
7.    System.out.println(formatter.format(date));
8. }
9. }
```

Output:

```
06/11/2017 12:26:18
```

Get Current Date: java.time.LocalDate

The LocalDate.now() method returns the instance of LocalDate class. If we print the instance of LocalDate class, it prints current date.

```
1. System.out.println(java.time.LocalDate.now());
```

Output:

```
2017-01-23
```


Get Current Time: java.time.LocalTime

The LocalTime.now() method returns the instance of LocalTime class. If we print the instance of LocalTime class, it prints current time.

```
1. System.out.println(java.time.LocalTime.now());
```

Output:

```
00:01:14.341
```

Get Current Date & Time: java.time.LocalDateTime

The LocalDateTime.now() method returns the instance of LocalDateTime class. If we print the instance of LocalDateTime class, it prints current date and time both.

```
1. System.out.println(java.time.LocalDateTime.now());
```

Output:

```
2017-01-24T00:03:31.593
```

Get Current Date & Time: java.time.Clock

The Clock.systemUTC().instant() method returns current date and time both.

```
1. System.out.println(java.time.Clock.systemUTC().instant());
```

Output:

```
2017-01-23T18:35:23.669Z
```

Get Current Date & Time: java.util.Date

By printing the instance of java.util.Date class, you can print current date and time in java. There are two ways to do so.

1st way:

```
1. java.util.Date date=new java.util.Date();
2. System.out.println(date);
```

2nd way:

```
1. long millis=System.currentTimeMillis();
2. java.util.Date date=new java.util.Date(millis);
3. System.out.println(date);
```

Output:


```
Thu Mar 26 08:22:02 IST 2015
```

Get Current Date: java.sql.Date

By printing the instance of java.sql.Date class, you can print current date in java. It doesn't print time. This date instance is generally used to save current date in database.

```
1. long millis=System.currentTimeMillis();
2. java.sql.Date date=new java.sql.Date(millis);
3. System.out.println(date);
```

Output:

```
2015-03-26
```

Get Current Date & Time: java.util.Calendar

Calendar class can be used to get the instance of Date class. The getTime() method of Calendar class returns the instance of java.util.Date. The Calendar.getInstance() method returns the instance of Calendar class.

```
1. Date date=java.util.Calendar.getInstance().getTime();
2. System.out.println(date);
```

Output:

```
Thu Mar 26 08:22:02 IST 2015
```

Methods

Creating Method

Considering the following example to explain the syntax of a method −

Syntax

```
public static int methodName(int a, int b) {
  // body
}
```

Here,

- **public static** − modifier
- **int** − return type
- **methodName** − name of the method
- **a, b** − formal parameters
- **int a, int b** − list of parameters

Method definition consists of a method header and a method body. The same is shown in the following syntax −

Syntax

```
modifier returnType nameOfMethod (Parameter List) {
  // method body
}
```

The syntax shown above includes −

- **modifier** − It defines the access type of the method and it is optional to use.
- **returnType** − Method may return a value.
- **nameOfMethod** − This is the method name. The method signature consists of the method name and the parameter list.
- **Parameter List** − The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- **method body** − The method body defines what the method does with the statements.

Example

Here is the source code of the above defined method called **min()**. This method takes two parameters num1 and num2 and returns the maximum between the two −

```
/** the snippet returns the minimum between two numbers */


public static int minFunction(int n1, int n2) {

  int min;
```

```
  if (n1 > n2)
    min = n2;
  else
    min = n1;

  return min;
}
```


Method Calling

For using a method, it should be called. There are two ways in which a method is called i.e., method returns a value or returning nothing (no return value).

The process of method calling is simple. When a program invokes a method, the program control gets transferred to the called method. This called method then returns control to the caller in two conditions, when −

- the return statement is executed.
- it reaches the method ending closing brace.

The methods returning void is considered as call to a statement. Lets consider an example −

```
System.out.println("This is tutorialspoint.com!");
```

The method returning value can be understood by the following example −

```
int result = sum(6, 9);
```

Example

```
public class ExampleMinNumber {


  public static void main(String[] args) {
    int a = 11;
    int b = 6;
    int c = minFunction(a, b);
    System.out.println("Minimum Value = " + c);
  }
```

```
  /** returns the minimum of two numbers */
  public static int minFunction(int n1, int n2) {
    int min;
    if (n1 > n2)
      min = n2;
    else
      min = n1;

    return min;
  }
}
```

This will produce the following result −

Output

```
Minimum value = 6
```

The void Keyword

Example

```
public class ExampleVoid {

  public static void main(String[] args) {
    methodRankPoints(255.7);
  }

  public static void methodRankPoints(double points) {
    if (points >= 202.5) {
      System.out.println("Rank:A1");
    }else if (points >= 122.4) {
```

```
      System.out.println("Rank:A2");
    }else {
      System.out.println("Rank:A3");
    }
  }
}
```


This will produce the following result −

Output

```
Rank:A1
```

Passing Parameters by Value

Example

The following program shows an example of passing parameter by value. The values of the arguments remains the same even after the method invocation.

```
public class swappingExample {

  public static void main(String[] args) {
    int a = 30;
    int b = 45;
    System.out.println("Before swapping, a = " + a + " and b = " + b);

    // Invoke the swap method
    swapFunction(a, b);
    System.out.println("\n**Now, Before and After swapping values will be same here**:");
    System.out.println("After swapping, a = " + a + " and b is " + b);
  }

  public static void swapFunction(int a, int b) {
```

```
    System.out.println("Before swapping(Inside), a = " + a + " b = " + b);


    // Swap n1 with n2
    int c = a;
    a = b;
    b = c;
    System.out.println("After swapping(Inside), a = " + a + " b = " + b);
  }
}
```


This will produce the following result −

Output

```
Before swapping, a = 30 and b = 45
Before swapping(Inside), a = 30 b = 45
After swapping(Inside), a = 45 b = 30

**Now, Before and After swapping values will be same here**:
After swapping, a = 30 and b is 45
```

Method overloading

If two or more method in a class have same name but different parameters, it is known as method overloading. Overloading always occur in the same class(unlike method overriding).

Method overloading is one of the ways through which java supports polymorphism. Method overloading can be done by changing number of arguments or by changing the data type of arguments. If two or more method have same name and same parameter list **but differs in return type are not** said to be overloaded method

Note: Overloaded method can have different access modifiers.

Different ways of Method overloading

There are two different ways of method overloading

Method overloading by changing data type of Arguments

Example :

```
class Calculate
{
```

```
void sum (int a, int b)
{
 System.out.println("sum is"+(a+b)) ;
}
void sum (float a, float b)
{
 System.out.println("sum is"+(a+b));
}
Public static void main (String[] args)
{
 Calculate  cal = new Calculate();
 cal.sum (8,5);     //sum(int a, int b) is method is called.
 cal.sum (4.6f, 3.8f); //sum(float a, float b) is called.
}
}
```

```
Sum is 13
Sum is 8.4
```

You can see that sum() method is overloaded two times. The first takes two integer arguments, the second takes two float arguments.

Method overloading by changing no. of argument.

Example :

```
class Area
{
 void find(int l, int b)
 {
  System.out.println("Area is"+(l*b)) ;
 }
 void find(int l, int b,int h)
 {
  System.out.println("Area is"+(l*b*h));
 }
```

```
public static void main (String[] args)
{
 Area  ar = new Area();
 ar.find(8,5);     //find(int l, int b) is method is called.
 ar.find(4,6,2);    //find(int l, int b,int h) is called.
}
}
```



```
Area is 40
Area is 48
```

In this example the find() method is overloaded twice. The first takes two arguments to calculate area, and the second takes three arguments to calculate area.

When an overloaded method is called java look for match between the arguments to call the method and the method's parameters. This match need not always be exact, sometime when exact match is not found, Java automatic type conversion plays a vital role.

Example of Method overloading with type promotion.

```
class Area
{
 void find(long l,long b)
 {
  System.out.println("Area is"+(l*b)) ;
 }
 void find(int l, int b,int h)
 {
  System.out.println("Area is"+(l*b*h));
 }
 public static void main (String[] args)
 {
  Area  ar = new Area();
  ar.find(8,5);     //automatic type conversion from find(int,int) to find(long,long) .
  ar.find(2,4,6)    //find(int l, int b,int h) is called.
 }
```

```
}
```

```
Area is 40
Area is 48
```

Method Overriding

When a method in a sub class has same name, same number of arguments and same type signature as a method in its super class, then the method is known as overridden method. Method overriding is also referred to as runtime polymorphism. The key benefit of overriding is the abitility to **define method that's specific to a particular subclass type**.

Example of Method Overriding

```
class Animal
{
   public void eat()
   {
      System.out.println("Generic Animal eating");
   }
}

class Dog extends Animal
{
   public void eat()   //eat() method overriden by Dog class.
   {
      System.out.println("Dog eat meat");
   }
}
```

As you can see here Dog class gives it own implementation of `eat()` method. Method must have same name and same type signature.

NOTE : Static methods cannot be overridden because, a static method is bounded with class where as instance method is bounded with object.

Covariant return type

Since Java 5, it is possible to override a method by changing its return type. If subclass override any method by changing the return type of super class method, then the return type of overriden method must be **subtype of return type** declared in original method inside the super class. This is the only way by which method can be overriden by changing its return type.

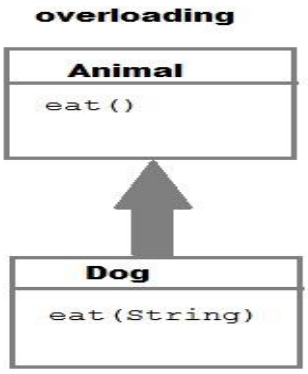
Example :

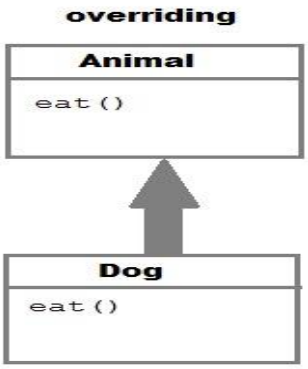

```
class Animal
{
   Animal myType()
   {
      return new Animal();
   }
}

class Dog extends Animal
{
   Dog myType()    //Legal override after Java5 onward
   {
      return new Dog();
   }
}
```

Difference between Overloading and Overriding

Method Overloading	**Method Overriding**
Parameter must be different and name must be same.	Both name and parameter must be same.
Compile time polymorphism.	Runtime polymorphism.

Method Overloading	Method Overriding
Increase readability of code.	Increase reusability of code.
Access specifier can be changed.	Access specifier cannot be more restrictive than original method(can be less restrictive).

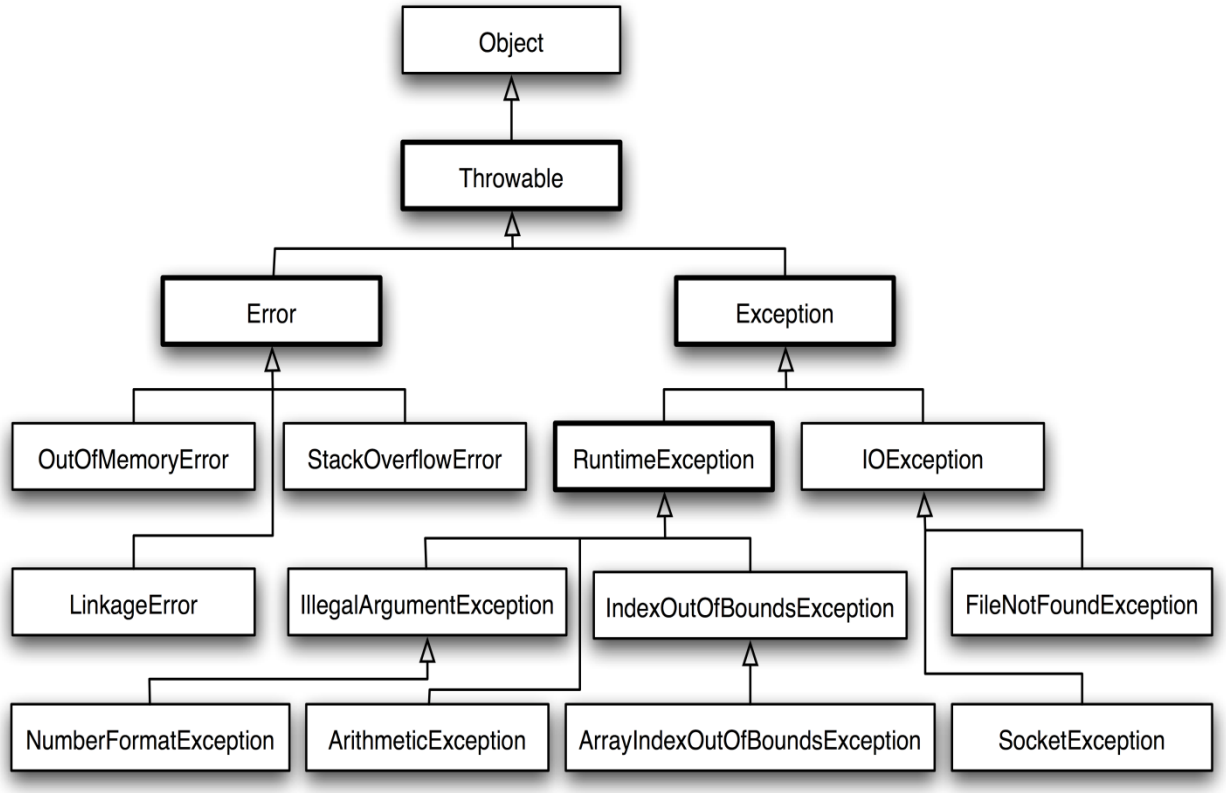




Exceptions

Checked exceptions − A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.

Unchecked exceptions − An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.



Multiple catch blocks in Java

Exception Handling example programs.

```
catch(Exception e){
  //This catch block catches all the exceptions
}
```

Example of Multiple catch blocks

```
class Example2{
  public static void main(String args[]){
    try{
       int a[]=new int[7];
       a[4]=30/0;
       System.out.println("First print statement in try block");
    }
    catch(ArithmeticException e){
      System.out.println("Warning: ArithmeticException");
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println("Warning: ArrayIndexOutOfBoundsException");
    }
    catch(Exception e){
      System.out.println("Warning: Some Other exception");
    }
```

```
  System.out.println("Out of try-catch block...");
 }
}
```

Output:

```
Warning: ArithmeticException
Out of try-catch block...
```

Throw Example

To understand this example you should know what is throw keyword and how it works, refer this guide: throw keyword in java.

```
public class Example1{
  void checkAge(int age){
            if(age<18)
              throw new ArithmeticException("Not Eligible for voting");
            else
              System.out.println("Eligible for voting");
  }
  public static void main(String args[]){
            Example1 obj = new Example1();
            obj.checkAge(13);
            System.out.println("End Of Program");
  }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException:
Not Eligible for voting
at Example1.checkAge(Example1.java:4)
at Example1.main(Example1.java:10)
```

Throws Example

To understand this example you should know what is throws clause and how it is used in method declaration for exception handling, refer this guide: throws in java.

```
public class Example1{
  int division(int a, int b) throws ArithmeticException{
            int t = a/b;
            return t;
  }
  public static void main(String args[]){
            Example1 obj = new Example1();
            try{
```

```
            System.out.println(obj.division(15,0));
          }
          catch(ArithmeticException e){
            System.out.println("You shouldn't divide number by zero");
          }
  }
}
```

Output:

```
You shouldn't divide number by zero
```

- **Arithmetic exception**

```
// Java program to demonstrate ArithmeticException

class ArithmeticException_Demo

{

   public static void main(String args[])

   {

      try {

         int a = 30, b = 0;

         int c = a/b;  // cannot divide by zero

         System.out.println ("Result = " + c);

      }

      catch(ArithmeticException e) {

         System.out.println ("Can't divide a number by 0");

      }

   }

}
```

- Run on IDE
- **Output:**
- Can't divide a number by 0
- **NullPointer Exception**

```
//Java program to demonstrate NullPointerException
class NullPointer_Demo
{
   public static void main(String args[])
   {
      try {
         String a = null; //null value
         System.out.println(a.charAt(0));
      } catch(NullPointerException e) {
         System.out.println("NullPointerException..");
      }
   }
}
```

- Run on IDE
- **Output:**
- NullPointerException..

- **StringIndexOutOfBound Exception**

```
// Java program to demonstrate StringIndexOutOfBoundsException
class StringIndexOutOfBound_Demo
{
   public static void main(String args[])
   {
      try {
         String a = "This is like chipping "; // length is 22
         char c = a.charAt(24); // accessing 25th element
         System.out.println(c);
      }
```

```
        catch(StringIndexOutOfBoundsException e) {
            System.out.println("StringIndexOutOfBoundsException");
        }
    }
}
```

- Run on IDE
- **Output:**
- StringIndexOutOfBoundsException
- **FileNotFound Exception**

```
//Java program to demonstrate FileNotFoundException

import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileReader;

 class File_notFound_Demo {


    public static void main(String args[])  {

        try {


            // Following file does not exist

            File file = new File("E://file.txt");


            FileReader fr = new FileReader(file);

        } catch (FileNotFoundException e) {

           System.out.println("File does not exist");

        }

    }

}
```

- Run on IDE
- **Output:**
- File does not exist

- **NumberFormat Exception**

```
// Java program to demonstrate NumberFormatException

class  NumberFormat_Demo

{

   public static void main(String args[])

   {

      try {

         // "akki" is not a number

         int num = Integer.parseInt ("akki") ;



         System.out.println(num);

      } catch(NumberFormatException e) {

         System.out.println("Number format exception");

      }

   }

}
```

- Run on IDE
- **Output:**
- Number format exception

- **ArrayIndexOutOfBounds Exception**

```
// Java program to demonstrate ArrayIndexOutOfBoundException

class ArrayIndexOutOfBound_Demo

{
```

```
    public static void main(String args[])
    {
      try{
        int a[] = new int[5];
        a[6] = 9; // accessing 7th element in an array of
                  // size 5
      }
      catch(ArrayIndexOutOfBoundsException e){
        System.out.println ("Array Index is Out Of Bounds");
      }
    }
}
```

- Run on IDE
- **Output:**

```
Array Index is Out Of Bounds
```

User-Defined Exceptions

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, user can also create exceptions which are called 'user-defined Exceptions'. Following steps are followed for the creation of user-defined Exception.

- The user should create an exception class as a subclass of Exception class. Since all the exceptions are subclasses of Exception class, the user should also make his class a subclass of it. This is done as:

```
class MyException extends Exception
```

- We can write a default constructor in his own exception class.

```
MyException(){}
```

- We can also create a parameterized constructor with a string as a parameter. We can use this to store exception details. We can call super class(Exception) constructor from this and send the string there.

```
MyException(String str)
{
```

- super(str);
- }

- To raise exception of user-defined type, we need to create an object to his exception class and throw it using throw clause, as:

- MyException me = new MyException("Exception details");

throw me;

- The following program illustrates how to create own exception class MyException.
- Details of account numbers, customer names, and balance amounts are taken in the form of three arrays.
- In main() method, the details are displayed using a for-loop. At this time, check is done if in any account the balance amount is less than the minimum balance amount to be ept in the account.
- If it is so, then MyException is raised and a message is displayed "Balance amount is less".

```
// Java program to demonstrate user defined exception


// This program throws an exception whenever balance

// amount is below Rs 1000

class MyException extends Exception

{

   //store account information

   private static int accno[] = {1001, 1002, 1003, 1004};


   private static String name[] =

           {"Nish", "Shubh", "Sush", "Abhi", "Akash"};


   private static double bal[] =

      {10000.00, 12000.00, 5600.0, 999.00, 1100.55};


   // default constructor
```

```
MyException() {    }

// parametrized constructor
MyException(String str) { super(str); }

// write main()
public static void main(String[] args)
{
   try  {
      // display the heading for the table
      System.out.println("ACCNO" + "\t" + "CUSTOMER" +
                              "\t" + "BALANCE");

      // display the actual account information
      for (int i = 0; i < 5 ; i++)
      {
         System.out.println(accno[i] + "\t" + name[i] +
                                 "\t" + bal[i]);

         // display own exception if balance < 1000
         if (bal[i] < 1000)
         {
            MyException me =
              new MyException("Balance is less than 1000");
            throw me;
         }
```

```
        }
      } //end of try


      catch (MyException e) {
         e.printStackTrace();
      }
   }
}
```

Run on IDE

RunTime Error

```
MyException: Balance is less than 1000
  at MyException.main(fileProperty.java:36)
```

Output:

```
ACCNO   CUSTOMER   BALANCE
1001   Nish   10000.0
1002   Shubh   12000.0
1003   Sush   5600.0
1004   Abhi   999.0
```

Java - Files and I/O

Create File

How to create a file in java

Read File

1) How to read a file in java using BufferedInputStream
2) How to read a file in java using BufferedReader

Write/Append File

1) How to write to a file in java using FileOutputStream
2) How to write to file in Java using BufferedWriter
3) Append to a file in java using BufferedWriter, PrintWriter, FileWriter

Delete/Rename File

1) How to delete file in Java using delete() Method
2) How to rename file in Java using renameTo() method

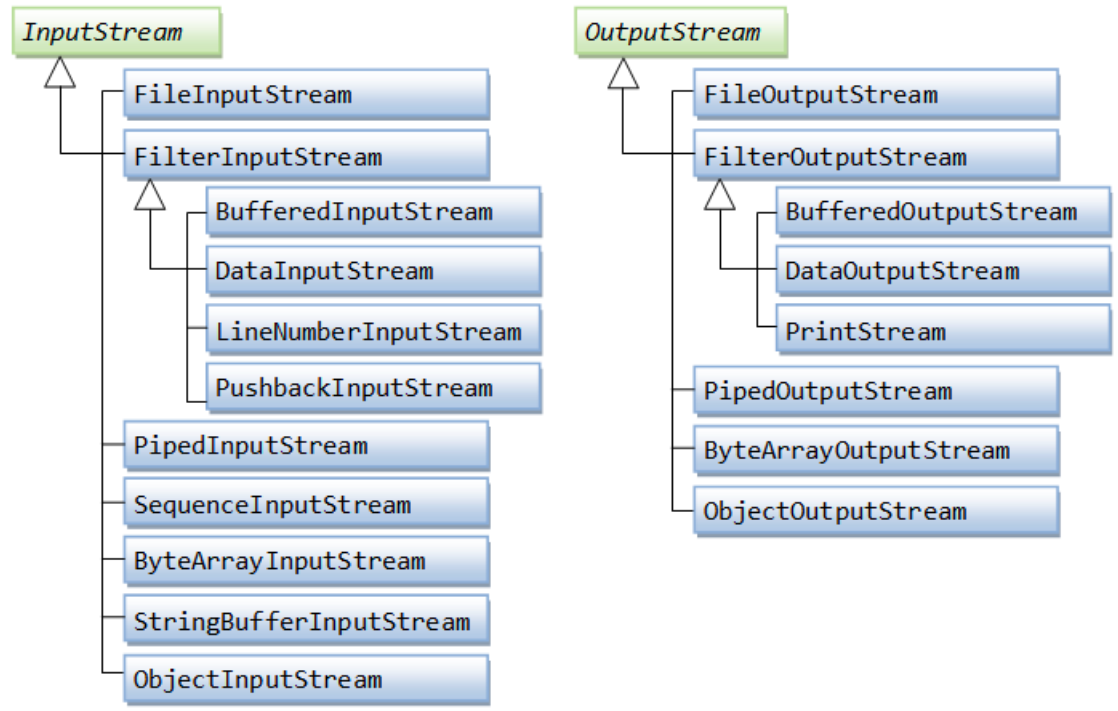
File compression

How to compress a File in GZip format

Misc

1) How to Copy a File to another File in Java
2) How to get the last modified date of a file in java
3) How to make a File Read Only in Java
4) How to check if a File is hidden

- **Standard Input** − This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
- **Standard Output** − This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.
- **Standard Error** − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.



What is a Java Thread?

A thread is actually a lightweight process. Unlike many other computer languages, Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run**concurrently**. Each part of such a program is called thread and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

Next concept in this Java Thread blog is integral to the concept Threads and Multithreading.

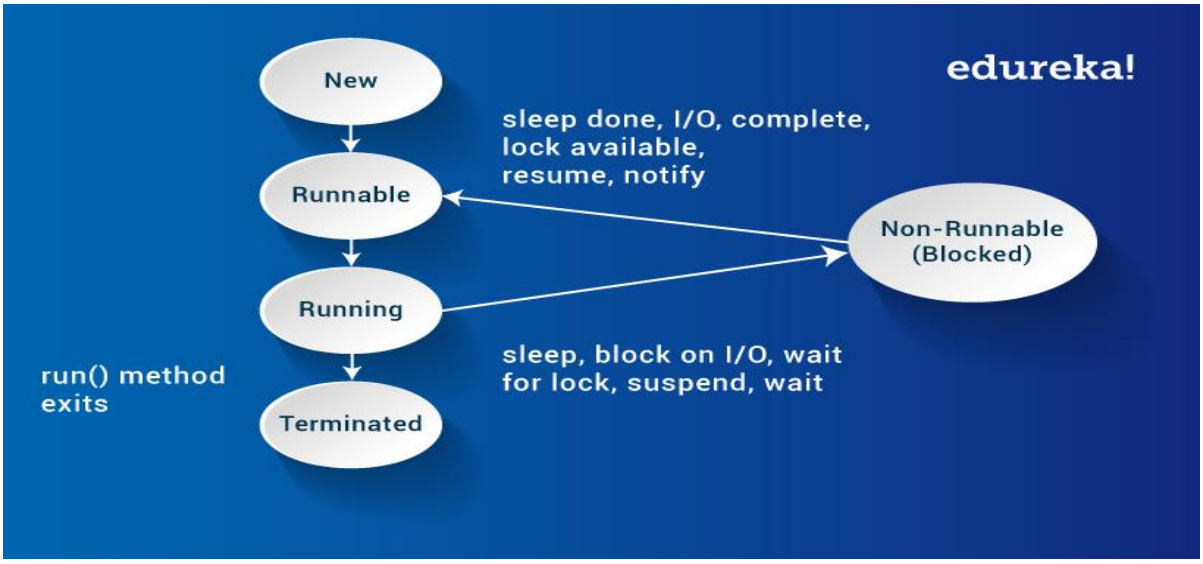
The Java Thread Model

The Java run-time system depends on threads for many things. Threads reduce inefficiency by preventing the waste of CPU cycles.

Threads exist in several states. Following are those states:

- **New** – When we create an instance of Thread class, a thread is in a new state.
- **Running –** The Java thread is in running state.
- **Suspended** – A running thread can be **suspended**, which temporarily suspends its activity. A suspended thread can then be resumed, allowing it to pick up where it left off.
- **Blocked** – A java thread can be blocked when waiting for a resource.

- **Terminated** – A thread can be terminated, which halts its execution immediately at any given time. Once a thread is terminated, it cannot be resumed.



So, this was all about the Java Thread states. Now, let us jump to most important topic of Java threads i.e. thread class and runnable interface. We will discuss these one by one below.

Multithreading in Java : Thread Class and Runnable Interface

Java's multithreading system is built upon the Thread class, its methods, and its companion interface,**Runnable**. To create a new thread, your program will either extend **Thread** or **implement** the **Runnableinterface**.

The Thread class defines several methods that help manage threads.The table below displays the same:

Method	Meaning
getName	Obtain thread's name
getPriority	Obtain thread's priority
isAlive	Determine if a thread is still running
join	Wait for a thread to terminate
run	Entry point for the thread
sleep	Suspend a thread for a period of time
start	Start a thread by calling its run method

Now let us see how to use a Thread which begins with the **main java thread**, that all Java programs have.

Main Java Thread

Now let us see how to use Thread and Runnable interface to create and manage threads, beginning with the**main java thread,** that all Java programs have. So, let us discuss the main thread.

Why is Main Thread so important?

- Because this thread effects the other 'child' threads
- Because it performs various shutdown actions
- It is created automatically when your program is started.

So, this was the main thread. Let's see how we can create a java thread?

How to Create a Java Thread?

Java lets you create thread in following two ways:-

- By **implementing** the **Runnable interface**.
- By **extending** the **Thread**

Let's see how both the ways help in implementing Java thread.

Runnable Interface

The easiest way to create a thread is to create a class that implements the **Runnable** interface.

To implement Runnable interface, a class need only implement a single method called run( ), which is declared like this:

```
public void run( )
```

Inside run( ), we will define the code that constitutes the new thread

Example:

```
public class MyClass implements Runnable {

public void run(){

System.out.println("MyClass running");

  }

}
```

To execute the run() method by a thread, pass an instance of MyClass to a Thread in its constructor(*A **constructor in Java** is a block of code similar to a method that's called when an instance of an object is created*). Here is how that is done:

```
Thread t1 = new Thread(new MyClass ());

t1.start();
```

When the thread is started it will call the run() method of the MyClass instance instead of executing its own run() method. The above example would print out the text "**MyClass running**".

Extending Java Thread

The second way to create a thread is to create a new class that extends Thread, then override the run() method and then to create an instance of that class. The run() method is what is executed by the thread after you call start(). Here is an example of creating a Java Thread subclass:

```
public class MyClass extends Thread {

    public void run(){

    System.out.println("MyClass running");

  }

}
```

To create and start the above thread you can do like this:

```
MyClass t1 = new MyClass ();

T1.start();
```

When the run() method executes it will print out the text "**MyClass running**".

*So far, we have been using only two threads: the **main** thread and one **child** thread. However, our program can affect as many threads as it needs. Let's see how we can create multiple threads.*

Creating Multiple Threads

```
 class MyThread implements Runnable {

 String name;

 Thread t;

    MyThread String thread){
```

```
    name = threadname;
    t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start();
}

public void run() {
 try {
     for(int i = 5; i > 0; i--) {
     System.out.println(name + ": " + i);
      Thread.sleep(1000);
}
}catch (InterruptedException e) {
     System.out.println(name + "Interrupted");
}
     System.out.println(name + " exiting.");
}
}

class MultiThread {
public static void main(String args[]) {
     new MyThread("One");
     new MyThread("Two");
     new NewThread("Three");
try {
```

```
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
            System.out.println("Main thread exiting.");
            }
        }
```

The output from this program is shown here:

```
New thread: Thread[One,5,main]
 New thread: Thread[Two,5,main]
 New thread: Thread[Three,5,main]
 One: 5
 Two: 5
 Three: 5
 One: 4
 Two: 4
 Three: 4
 One: 3
 Three: 3
 Two: 3
 One: 2
 Three: 2
 Two: 2
 One: 1
 Three: 1
 Two: 1
 One exiting.
 Two exiting.
```

```
Three exiting.
Main thread exiting.
```

Synchronization in Java

Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. static synchronization.
2. Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

1. by synchronized method

2. by synchronized block
3. by static synchronization

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks contains several lock implementations.

Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

```
1. class Table{
2. void printTable(int n){//method not synchronized
3.   for(int i=1;i<=5;i++){
4.     System.out.println(n*i);
5.     try{
6.      Thread.sleep(400);
7.     }catch(Exception e){System.out.println(e);}
8.   }
9.
10.  }
11. }
12.
13. class MyThread1 extends Thread{
14. Table t;
15. MyThread1(Table t){
16. this.t=t;
17. }
18. public void run(){
19. t.printTable(5);
20. }
21.
22. }
23. class MyThread2 extends Thread{
24. Table t;
25. MyThread2(Table t){
26. this.t=t;
27. }
28. public void run(){
29. t.printTable(100);
```

```
30. }
31. }
32.
33. class TestSynchronization1{
34. public static void main(String args[]){
35. Table obj = new Table();//only one object
36. MyThread1 t1=new MyThread1(obj);
37. MyThread2 t2=new MyThread2(obj);
38. t1.start();
39. t2.start();
40. }
41. }
```

```
Output: 5
      100
      10
      200
      15
      300
      20
      400
      25
      500
```

Java synchronized method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

```
1.  //example of java synchronized method
2.  class Table{
3.   synchronized void printTable(int n){//synchronized method
4.     for(int i=1;i<=5;i++){
5.       System.out.println(n*i);
6.       try{
7.        Thread.sleep(400);
8.       }catch(Exception e){System.out.println(e);}
9.     }
10.
11.  }
12. }
13.
```

```
14. class MyThread1 extends Thread{
15. Table t;
16. MyThread1(Table t){
17. this.t=t;
18. }
19. public void run(){
20. t.printTable(5);
21. }
22.
23. }
24. class MyThread2 extends Thread{
25. Table t;
26. MyThread2(Table t){
27. this.t=t;
28. }
29. public void run(){
30. t.printTable(100);
31. }
32. }
33.
34. public class TestSynchronization2{
35. public static void main(String args[]){
36. Table obj = new Table();//only one object
37. MyThread1 t1=new MyThread1(obj);
38. MyThread2 t2=new MyThread2(obj);
39. t1.start();
40. t2.start();
41. }
42. }
```

```
Output: 5
    10
    15
    20
    25
    100
    200
    300
    400
    500
```

Example of synchronized method by using annonymous class

In this program, we have created the two threads by annonymous class, so less coding is required.

```
1.  //Program of synchronized method by using annonymous class
2.  class Table{
3.   synchronized void printTable(int n){//synchronized method
4.     for(int i=1;i<=5;i++){
5.       System.out.println(n*i);
6.       try{
7.        Thread.sleep(400);
8.       }catch(Exception e){System.out.println(e);}
9.     }
10.
11.  }
12. }
13.
14. public class TestSynchronization3{
15. public static void main(String args[]){
16. final Table obj = new Table();//only one object
17.
18. Thread t1=new Thread(){
19. public void run(){
20. obj.printTable(5);
21. }
22. };
23. Thread t2=new Thread(){
24. public void run(){
25. obj.printTable(100);
26. }
27. };
28.
29. t1.start();
30. t2.start();
31. }
32. }
```

```
Output: 5
        10
        15
        20
        25
        100
        200
        300
        400
        500
```

notify() and wait() examples

notify() and wait() - example 1

```
public class ThreadA {
   public static void main(String[] args){
      ThreadB b = new ThreadB();
      b.start();

      synchronized(b){
         try{
            System.out.println("Waiting for b to complete...");
            b.wait();
         }catch(InterruptedException e){
            e.printStackTrace();
         }

         System.out.println("Total is: " + b.total);
      }
   }
}

class ThreadB extends Thread{
   int total;
   @Override
   public void run(){
      synchronized(this){
         for(int i=0; i<100 ; i++){
            total += i;
         }
         notify();
      }
   }
}
```

In the example above, an object, b, is synchronized. b completes the calculation before Main thread outputs its total value.

Output:

```
Waiting for b to complete...
Total is: 4950
```

If b is not synchonized like the code below:

```
public class ThreadA {
        public static void main(String[] args) {
                ThreadB b = new ThreadB();
                b.start();

                System.out.println("Total is: " + b.total);

        }
}
```

```
class ThreadB extends Thread {
        int total;

        @Override
        public void run() {
                for (int i = 0; i < 100; i++) {
                        total += i;
                }
        }
}
```

The result would be 0, 10, etc. Because sum is not finished before it is used.

3. notify() and wait() - example 2

The second example is more complex, see the comments.

```
import java.util.Vector;

class Producer extends Thread {

   static final int MAXQUEUE = 5;
   private Vector messages = new Vector();

   @Override
   public void run() {
      try {
         while (true) {
            putMessage();
            //sleep(5000);
         }
      } catch (InterruptedException e) {
      }
   }

   private synchronized void putMessage() throws InterruptedException {
      while (messages.size() == MAXQUEUE) {
         wait();
      }
      messages.addElement(new java.util.Date().toString());
      System.out.println("put message");
      notify();
   }

   // Called by Consumer
   public synchronized String getMessage() throws InterruptedException {
      notify();
      while (messages.size() == 0) {
         wait();
      }
```

```
        String message = (String) messages.firstElement();
        messages.removeElement(message);
        return message;
    }
}

class Consumer extends Thread {

    Producer producer;

    Consumer(Producer p) {
        producer = p;
    }

    @Override
    public void run() {
        try {
            while (true) {
                String message = producer.getMessage();
                System.out.println("Got message: " + message);
                //sleep(200);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String args[]) {
        Producer producer = new Producer();
        producer.start();
        new Consumer(producer).start();
    }
}
```

A possible output sequence:

```
Got message: Fri Dec 02 21:37:21 EST 2011
put message
put message
put message
put message
put message
Got message: Fri Dec 02 21:37:21 EST 2011
Got message: Fri Dec 02 21:37:21 EST 2011
Got message: Fri Dec 02 21:37:21 EST 2011
Got message: Fri Dec 02 21:37:21 EST 2011
Got message: Fri Dec 02 21:37:21 EST 2011
put message
```

```
put message
put message
put message
put message
Got message: Fri Dec 02 21:37:21 EST 2011
Got message: Fri Dec 02 21:37:21 EST 2011
Got message: Fri Dec 02 21:37:21 EST 2011
```

Java - Inheritance

In simple words, Inheritance is way to define new a class, using classes which have already been defined.

extends Keyword

extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

Syntax

```
class Super {
   .....
   .....
}
class Sub extends Super {
   .....
   .....
}
```

Sample Code

Following is an example demonstrating Java inheritance. In this example, you can observe two classes namely Calculation and My_Calculation.

Using extends keyword, the My_Calculation inherits the methods addition() and Subtraction() of Calculation class.

Copy and paste the following program in a file with name My_Calculation.java

Example

```
class Calculation {

   int z;
```

```
  public void addition(int x, int y) {
    z = x + y;
    System.out.println("The sum of the given numbers:"+z);
  }

  public void Subtraction(int x, int y) {
    z = x - y;
    System.out.println("The difference between the given numbers:"+z);
  }
}

public class My_Calculation extends Calculation {
  public void multiplication(int x, int y) {
    z = x * y;
    System.out.println("The product of the given numbers:"+z);
  }

  public static void main(String args[]) {
    int a = 20, b = 10;
    My_Calculation demo = new My_Calculation();
    demo.addition(a, b);
    demo.Subtraction(a, b);
    demo.multiplication(a, b);
  }
}
```

Compile and execute the above code as shown below.

```
javac My_Calculation.java
java My_Calculation
```

After executing the program, it will produce the following result –

Output

```
The sum of the given numbers:30
The difference between the given numbers:10
The product of the given numbers:200
```

The super keyword

The **super** keyword is similar to **this** keyword. Following are the scenarios where the super keyword is used.

- It is used to **differentiate the members** of superclass from the members of subclass, if they have same names.
- It is used to **invoke the superclass** constructor from subclass.
- **Example**

```
class Super_class {
   int num = 20;

   // display method of superclass
   public void display() {
      System.out.println("This is the display method of superclass");
   }
}

public class Sub_class extends Super_class {
   int num = 10;

   // display method of sub class
   public void display() {
      System.out.println("This is the display method of subclass");
   }

```

```
    public void my_method() {
      // Instantiating subclass
      Sub_class sub = new Sub_class();

      // Invoking the display() method of sub class
      sub.display();

      // Invoking the display() method of superclass
      super.display();

      // printing the value of variable num of subclass
      System.out.println("value of the variable named num in sub class:"+ sub.num);

      // printing the value of variable num of superclass
      System.out.println("value of the variable named num in super class:"+
   super.num);
    }

    public static void main(String args[]) {
      Sub_class obj = new Sub_class();
      obj.my_method();
    }
  }
```

- Compile and execute the above code using the following syntax.

```
javac Super_Demo
java Super
```

- On executing the program, you will get the following result –

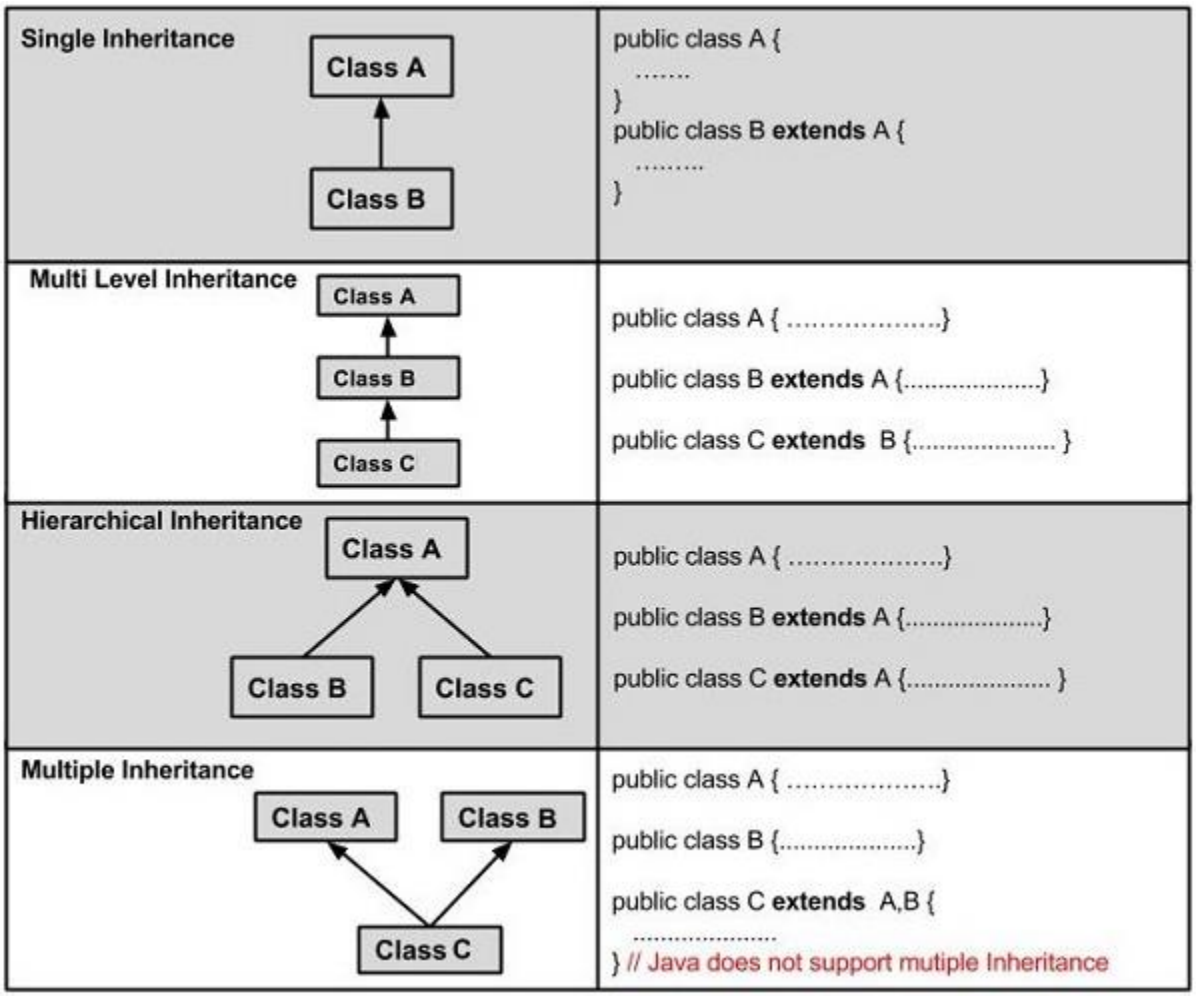
- **Output**

```
This is the display method of subclass
```

- This is the display method of superclass
- value of the variable named num in sub class:10
- value of the variable named num in super class:20

Types of Inheritance

There are various types of inheritance as demonstrated below.



Single Inheritance Example

File: TestInheritance.java

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class TestInheritance{
8. public static void main(String args[]){
9. Dog d=new Dog();
```

```
10. d.bark();
11. d.eat();
12. }}
```

Output:

```
barking...
eating...
```

Multilevel Inheritance Example

File: TestInheritance2.java

```
1.  class Animal{
2.  void eat(){System.out.println("eating...");}
3.  }
4.  class Dog extends Animal{
5.  void bark(){System.out.println("barking...");}
6.  }
7.  class BabyDog extends Dog{
8.  void weep(){System.out.println("weeping...");}
9.  }
10. class TestInheritance2{
11. public static void main(String args[]){
12. BabyDog d=new BabyDog();
13. d.weep();
14. d.bark();
15. d.eat();
16. }}
```

Output:

```
weeping...
barking...
eating...
```

Hierarchical Inheritance Example

File: TestInheritance3.java

```
1.  class Animal{
2.  void eat(){System.out.println("eating...");}
3.  }
4.  class Dog extends Animal{
5.  void bark(){System.out.println("barking...");}
6.  }
7.  class Cat extends Animal{
```

```
8. void meow(){System.out.println("meowing...");}
9. }
10. class TestInheritance3{
11. public static void main(String args[]){
12. Cat c=new Cat();
13. c.meow();
14. c.eat();
15. //c.bark();//C.T.Error
16. }}
```

Output:

```
meowing...
eating...
```

Super Keyword in Java

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
1. class Animal{
2. String color="white";
3. }
4. class Dog extends Animal{
5. String color="black";
6. void printColor(){
7. System.out.println(color);//prints color of Dog class
8. System.out.println(super.color);//prints color of Animal class
9. }
10. }
11. class TestSuper1{
```

```
12. public static void main(String args[]){
13. Dog d=new Dog();
14. d.printColor();
15. }}
```

Output:

```
black
white
```

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void eat(){System.out.println("eating bread...");}
6. void bark(){System.out.println("barking...");}
7. void work(){
8. super.eat();
9. bark();
10. }
11. }
12. class TestSuper2{
13. public static void main(String args[]){
14. Dog d=new Dog();
15. d.work();
16. }}
```

Output:

```
eating...
barking...
```

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

```
1. class Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. class Dog extends Animal{
5. Dog(){
6. super();
7. System.out.println("dog is created");
8. }
9. }
10. class TestSuper3{
11. public static void main(String args[]){
12. Dog d=new Dog();
13. }}
```

Output:

```
animal is created
dog is created
```

As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

Another example of super keyword where super() is provided by the compiler implicitly.

```
1. class Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. class Dog extends Animal{
5. Dog(){
6. System.out.println("dog is created");
7. }
8. }
9. class TestSuper4{
10. public static void main(String args[]){
11. Dog d=new Dog();
12. }}
```

Output:

```
animal is created
dog is created
```

super example: real use

Let's see the real use of super keyword. Here, Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are

using parent class constructor from child class. In such way, we are reusing the parent class constructor.

```
1. class Person{
2. int id;
3. String name;
4. Person(int id,String name){
5. this.id=id;
6. this.name=name;
7. }
8. }
9. class Emp extends Person{
10. float salary;
11. Emp(int id,String name,float salary){
12. super(id,name);//reusing parent constructor
13. this.salary=salary;
14. }
15. void display(){System.out.println(id+" "+name+" "+salary);}
16. }
17. class TestSuper5{
18. public static void main(String[] args){
19. Emp e1=new Emp(1,"ankit",45000f);
20. e1.display();
21. }}
```

Output:

```
1 ankit 45000
```

Polymorphism

What is Polymorphism in Java

The dictionary definition of *polymorphism* refers to a principle in biology in which an organism or species can have many different forms or stages. This principle can also be applied to object-oriented programming and languages like the Java language.

Example of Java Runtime Polymorphism

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
1. class Bike{
2.   void run(){System.out.println("running");}
3. }
4. class Splendor extends Bike{
5.   void run(){System.out.println("running safely with 60km");}
6.   public static void main(String args[]){
7.     Bike b = new Splendor();//upcasting
8.     b.run();
9.   }
10. }
```

Output:

```
running safely with 60km.
```

Java Runtime Polymorphism Example: Bank

Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.

Note: This example is also given in method overriding but there was no upcasting.

```
1. class Bank{
2. float getRateOfInterest(){return 0;}
3. }
4. class SBI extends Bank{
5. float getRateOfInterest(){return 8.4f;}
6. }
7. class ICICI extends Bank{
8. float getRateOfInterest(){return 7.3f;}
9. }
10. class AXIS extends Bank{
11. float getRateOfInterest(){return 9.7f;}
12. }
13. class TestPolymorphism{
14. public static void main(String args[]){
15. Bank b;
16. b=new SBI();
17. System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
18. b=new ICICI();
```

```
19. System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
20. b=new AXIS();
21. System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
22. }
23. }
```

Output:

```
SBI Rate of Interest: 8.4
ICICI Rate of Interest: 7.3
AXIS Rate of Interest: 9.7
```

Java Runtime Polymorphism Example: Shape

```
1.  class Shape{
2.  void draw(){System.out.println("drawing...");}
3.  }
4.  class Rectangle extends Shape{
5.  void draw(){System.out.println("drawing rectangle...");}
6.  }
7.  class Circle extends Shape{
8.  void draw(){System.out.println("drawing circle...");}
9.  }
10. class Triangle extends Shape{
11. void draw(){System.out.println("drawing triangle...");}
12. }
13. class TestPolymorphism2{
14. public static void main(String args[]){
15. Shape s;
16. s=new Rectangle();
17. s.draw();
18. s=new Circle();
19. s.draw();
20. s=new Triangle();
21. s.draw();
22. }
23. }
```

Output:

```
drawing rectangle...
drawing circle...
drawing triangle...
```

Java Runtime Polymorphism Example: Animal

```
1.  class Animal{
```

```
2.  void eat(){System.out.println("eating...");}
3.  }
4.  class Dog extends Animal{
5.  void eat(){System.out.println("eating bread...");}
6.  }
7.  class Cat extends Animal{
8.  void eat(){System.out.println("eating rat...");}
9.  }
10. class Lion extends Animal{
11. void eat(){System.out.println("eating meat...");}
12. }
13. class TestPolymorphism3{
14. public static void main(String[] args){
15. Animal a;
16. a=new Dog();
17. a.eat();
18. a=new Cat();
19. a.eat();
20. a=new Lion();
21. a.eat();
22. }}
```

Output:

```
eating bread...
eating rat...
eating meat...
```

Java Runtime Polymorphism with Data Member

A method is overridden, not the data members, so runtime polymorphism can't be achieved by data members.

In the example given below, both the classes have a data member speedlimit. We are accessing the data member by the reference variable of Parent class which refers to the subclass object. Since we are accessing the data member which is not overridden, hence it will access the data member of the Parent class always.

> *Rule: Runtime polymorphism can't be achieved by data members.*

```
1.  class Bike{
2.   int speedlimit=90;
3.  }
4.  class Honda3 extends Bike{
5.   int speedlimit=150;
6.
```

```
7.   public static void main(String args[]){
8.    Bike obj=new Honda3();
9.    System.out.println(obj.speedlimit);//90
10. }
```

Output:

```
90
```

Java Runtime Polymorphism with Multilevel Inheritance

Let's see the simple example of Runtime Polymorphism with multilevel inheritance.

```
1.  class Animal{
2.  void eat(){System.out.println("eating");}
3.  }
4.  class Dog extends Animal{
5.  void eat(){System.out.println("eating fruits");}
6.  }
7.  class BabyDog extends Dog{
8.  void eat(){System.out.println("drinking milk");}
9.  public static void main(String args[]){
10. Animal a1,a2,a3;
11. a1=new Animal();
12. a2=new Dog();
13. a3=new BabyDog();
14. a1.eat();
15. a2.eat();
16. a3.eat();
17. }
18. }
```

Output:

```
eating
eating fruits
drinking Milk
```

Try for Output

```
1.  class Animal{
2.  void eat(){System.out.println("animal is eating...");}
3.  }
4.  class Dog extends Animal{
5.  void eat(){System.out.println("dog is eating...");}
6.  }
7.  class BabyDog1 extends Dog{
```

```
8. public static void main(String args[]){
9. Animal a=new BabyDog1();
10. a.eat();
11. }}
```

Output:

```
Dog is eating
```

Abstraction

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Example of abstract class

```
1. abstract class A{}
```

Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of abstract method

```
1. abstract void printStatus();//no method body and abstract
2. abstract class Bank{
3. abstract int getRateOfInterest();
4. }
5. class SBI extends Bank{
6. int getRateOfInterest(){return 7;}
7. }
8. class PNB extends Bank{
9. int getRateOfInterest(){return 8;}
10. }
```

```
11.
12. class TestBank{
13. public static void main(String args[]){
14. Bank b;
15. b=new SBI();
16. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
17. b=new PNB();
18. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
19. }}
    ***
Rate of Interest is: 7 %
Rate of Interest is: 8 %
```

Interfaces

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

Syntax:

```
1. interface <interface_name>{
2.
3.     // declare constant fields
4.     // declare methods that abstract
5.     // by default.
6. }
```

Java Interface Example

In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

```
1. interface printable{
2. void print();
3. }
4. class A6 implements printable{
5. public void print(){System.out.println("Hello");}
6.
7. public static void main(String args[]){
8. A6 obj = new A6();
9. obj.print();
```

```
10.  }
11. }
```

Output:

```
Hello
```

Java Interface Example: Drawable

In this example, the Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

File: TestInterface1.java

```
1.  //Interface declaration: by first user
2.  interface Drawable{
3.  void draw();
4.  }
5.  //Implementation: by second user
6.  class Rectangle implements Drawable{
7.  public void draw(){System.out.println("drawing rectangle");}
8.  }
9.  class Circle implements Drawable{
10. public void draw(){System.out.println("drawing circle");}
11. }
12. //Using interface: by third user
13. class TestInterface1{
14. public static void main(String args[]){
15. Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()

16. d.draw();
17. }}
```

Output:

```
drawing circle
```

Java Interface Example: Bank

Let's see another example of java interface which provides the implementation of Bank interface.

File: TestInterface2.java

```
1. interface Bank{
2. float rateOfInterest();
3. }
4. class SBI implements Bank{
5. public float rateOfInterest(){return 9.15f;}
6. }
7. class PNB implements Bank{
8. public float rateOfInterest(){return 9.7f;}
9. }
10. class TestInterface2{
11. public static void main(String[] args){
12. Bank b=new SBI();
13. System.out.println("ROI: "+b.rateOfInterest());
14. }}
```

Output:

```
ROI: 9.15
```

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.

```
1. interface Printable{
2. void print();
3. }
4. interface Showable{
5. void show();
6. }
7. class A7 implements Printable,Showable{
8. public void print(){System.out.println("Hello");}
9. public void show(){System.out.println("Welcome");}
10.
11. public static void main(String args[]){
12. A7 obj = new A7();
13. obj.print();
14. obj.show();
15.  }
16. }
```

```
Output:Hello
     Welcome
```

***

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class**can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class**can be extended using keyword "extends".	An **interface class**can be implemented using keyword "implements".
8) A Java**abstract class**can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract class both.

```
1. //Creating interface that has 4 methods
2. interface A{
3. void a();//bydefault, public and abstract
4. void b();
5. void c();
6. void d();
7. }
8.
9. //Creating abstract class that provides the implementation of one method of A interface
10. abstract class B implements A{
11. public void c(){System.out.println("I am C");}
12. }
13.
14. //Creating subclass of abstract class, now we need to provide the implementation of rest of
    the methods
15. class M extends B{
16. public void a(){System.out.println("I am a");}
17. public void b(){System.out.println("I am b");}
18. public void d(){System.out.println("I am d");}
19. }
20.
21. //Creating a test class that calls the methods of A interface
22. class Test5{
23. public static void main(String args[]){
24. A a=new M();
25. a.a();
26. a.b();
27. a.c();
28. a.d();
29. }}
```

Output:

```
I am a
I am b
I am c
I am d
```

Encapsulation

Encapsulation in Java is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed of several medicines.

We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

The **Java Bean** class is the example of a fully encapsulated class.

Simple Example of Encapsulation in Java

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.

File: Student.java

```
1. //A Java class which is a fully encapsulated class.
2. //It has a private data member and getter and setter methods.
3. package com.javatpoint;
4. public class Student{
5. //private data member
6. private String name;
7. //getter method for name
8. public String getName(){
9. return name;
10. }
11. //setter method for name
12. public void setName(String name){
13. this.name=name
14. }
15. }
```

File: Test.java

```
1. //A Java class to test the encapsulated class.
2. package com.javatpoint;
3. class Test{
```

```
4. public static void main(String[] args){
5. //creating instance of the encapsulated class
6. Student s=new Student();
7. //setting value in the name member
8. s.setName("vijay");
9. //getting value of the name member
10. System.out.println(s.getName());
11. }
12. }
```

```
Compile By: javac -d . Test.java
Run By: java com.javatpoint.Test
```

Output:

```
vijay
```

Read-Only class

```
1. //A Java class which has only getter methods.
2. public class Student{
3. //private data member
4. private String college="AKG";
5. //getter method for college
6. public String getCollege(){
7. return college;
8. }
9. }
```

Now, you can't change the value of the college data member which is "AKG".

```
1. s.setCollege("KITE");//will render compile time error
```

Write-Only class

```
1. //A Java class which has only setter methods.
2. public class Student{
3. //private data member
4. private String college;
5. //getter method for college
6. public void setCollege(String college){
7. this.college=college;
8. }
9. }
```

Now, you can't get the value of the college, you can only change the value of college data member.

```
1. System.out.println(s.getCollege());//Compile Time Error, because there is no such method
2. System.out.println(s.college);//Compile Time Error, because the college data member is priv
   ate.
```

```
3. //So, it can't be accessed from outside the class
```

Another Example of Encapsulation in Java

Let's see another example of encapsulation that has only four fields with its setter and getter methods.

File: Account.java

```
1. //A Account class which is a fully encapsulated class.
2. //It has a private data member and getter and setter methods.
3. class Account {
4. //private data members
5. private long acc_no;
6. private String name,email;
7. private float amount;
8. //public getter and setter methods
9. public long getAcc_no() {
10.    return acc_no;
11. }
12. public void setAcc_no(long acc_no) {
13.    this.acc_no = acc_no;
14. }
15. public String getName() {
16.    return name;
17. }
18. public void setName(String name) {
19.    this.name = name;
20. }
21. public String getEmail() {
22.    return email;
23. }
24. public void setEmail(String email) {
25.    this.email = email;
26. }
27. public float getAmount() {
28.    return amount;
29. }
30. public void setAmount(float amount) {
31.    this.amount = amount;
32. }
33.
34. }
```

File: TestAccount.java

```
1. //A Java class to test the encapsulated class Account.
```

```
2. public class TestEncapsulation {
3. public static void main(String[] args) {
4.    //creating instance of Account class
5.    Account acc=new Account();
6.    //setting values through setter methods
7.    acc.setAcc_no(7560504000L);
8.    acc.setName("Sonoo Jaiswal");
9.    acc.setEmail("sonoojaiswal@.com");
10.   acc.setAmount(500000f);
11.   //getting values through getter methods
12.   System.out.println(acc.getAcc_no()+" "+acc.getName()+" "+acc.getEmail()+" "+acc.getAm
    ount());
13. }
14. }
```

Output:

```
7560504000 Sonoo Jaiswal sonoojaiswal@.com 500000.0
```

Using Jframe

Java JButton Example

```
1. import javax.swing.*;
2. public class ButtonExample {
3. public static void main(String[] args) {
4.    JFrame f=new JFrame("Button Example");
5.    JButton b=new JButton("Click Here");
6.    b.setBounds(50,100,95,30);
7.    f.add(b);
8.    f.setSize(400,400);
9.    f.setLayout(null);
10.   f.setVisible(true);
11. }
12. }
```

Output:

Java JLabel Example

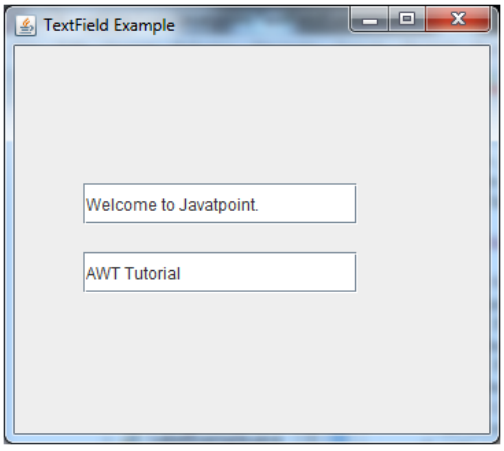
```
1. import javax.swing.*;
2. class LabelExample
3. {
4. public static void main(String args[])
```

```
5.    {
6.    JFrame f= new JFrame("Label Example");
7.    JLabel l1,l2;
8.    l1=new JLabel("First Label.");
9.    l1.setBounds(50,50, 100,30);
10.   l2=new JLabel("Second Label.");
11.   l2.setBounds(50,100, 100,30);
12.   f.add(l1); f.add(l2);
13.   f.setSize(300,300);
14.   f.setLayout(null);
15.   f.setVisible(true);
16.   }
17.   }
```

Output:

Java JTextField Example

```
1.  import javax.swing.*;
2.  class TextFieldExample
3.  {
4.  public static void main(String args[])
5.    {
6.    JFrame f= new JFrame("TextField Example");
7.    JTextField t1,t2;
8.    t1=new JTextField("Welcome to Javatpoint.");
9.    t1.setBounds(50,100, 200,30);
10.   t2=new JTextField("AWT Tutorial");
11.   t2.setBounds(50,150, 200,30);
12.   f.add(t1); f.add(t2);
13.   f.setSize(400,400);
14.   f.setLayout(null);
15.   f.setVisible(true);
16.   }
17.   }
```

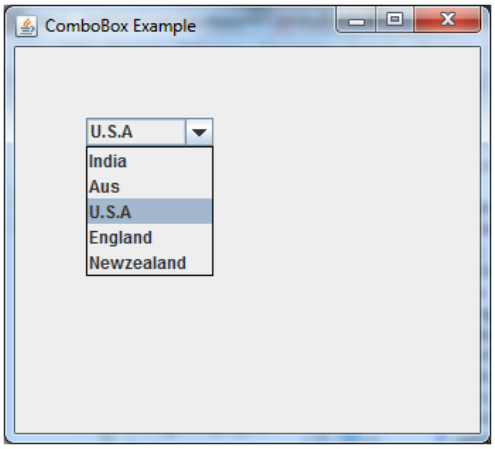


Output:

Java JComboBox Example

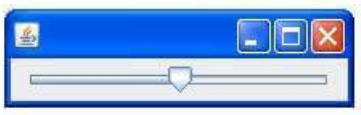
```
1.  import javax.swing.*;
2.  public class ComboBoxExample {
3.  JFrame f;
4.  ComboBoxExample(){
5.    f=new JFrame("ComboBox Example");
```

```
6.      String country[]={"India","Aus","U.S.A","England","Newzealand"};
7.      JComboBox cb=new JComboBox(country);
8.      cb.setBounds(50, 50,90,20);
9.      f.add(cb);
10.     f.setLayout(null);
11.     f.setSize(400,500);
12.     f.setVisible(true);
13. }
14. public static void main(String[] args) {
15.     new ComboBoxExample();
16. }
17. }
```



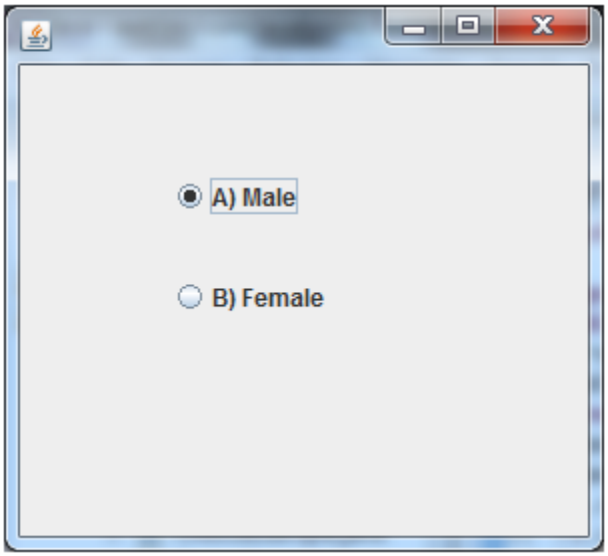
Java JSlider Example

```
1.  import javax.swing.*;
2.  public class SliderExample1 extends JFrame{
3.  public SliderExample1() {
4.  JSlider slider = new JSlider(JSlider.HORIZONTAL, 0, 50, 25);
5.  JPanel panel=new JPanel();
6.  panel.add(slider);
7.  add(panel);
8.  }
9.
10. public static void main(String s[]) {
11. SliderExample1 frame=new SliderExample1();
12. frame.pack();
13. frame.setVisible(true);
14. }
15. }
```



Java JRadioButton Example

```
1.  import javax.swing.*;
2.  public class RadioButtonExample {
3.  JFrame f;
4.  RadioButtonExample(){
5.  f=new JFrame();
6.  JRadioButton r1=new JRadioButton("A) Male");

7.  JRadioButton r2=new JRadioButton("B) Female")
    ;
8.  r1.setBounds(75,50,100,30);
9.  r2.setBounds(75,100,100,30);
10. ButtonGroup bg=new ButtonGroup();
11. bg.add(r1);bg.add(r2);
12. f.add(r1);f.add(r2);
13. f.setSize(300,300);
```



```
14. f.setLayout(null);
15. f.setVisible(true);
16. }
17. public static void main(String[] args) {
18.     new RadioButtonExample();
19. }
20. }
```

50+ Problem solved

Example 1: Program to print fibonacci series using for loop

```
public class JavaExample {

   public static void main(String[] args) {

      int count = 7, num1 = 0, num2 = 1;
      System.out.print("Fibonacci Series of "+count+" numbers:");

      for (int i = 1; i <= count; ++i)
      {
         System.out.print(num1+" ");
         int sumOfPrevTwo = num1 + num2;
         num1 = num2;
         num2 = sumOfPrevTwo;
      }
   }
}
```

Output:

```
Fibonacci Series of 7 numbers:0 1 1 2 3 5 8
```

Example: Finding factorial using for loop

```
public class JavaExample {
```

```
  public static void main(String[] args) {

            //We will find the factorial of this number
     int number = 5;
     long fact = 1;
     for(int i = 1; i <= number; i++)
     {
        fact = fact * i;
     }
     System.out.println("Factorial of "+number+" is: "+fact);
  }
}
```

Output:

```
Factorial of 5 is: 120
```

Example: Swapping two numbers using bitwise operator

```
import java.util.Scanner;
public class JavaExample
{
   public static void main(String args[])
   {
      int num1, num2;
      Scanner scanner = new Scanner(System.in);
      System.out.print("Enter first number:");
      num1 = scanner.nextInt();
      System.out.print("Enter second number:");
      num2 = scanner.nextInt();

      //num1 becomes 1111 = 15
      num1 = num1 ^ num2;
      //num2 becomes 1010 = 10
      num2 = num1 ^ num2;
      //num1 becomes 0101 = 5
      num1 = num1 ^ num2;
      scanner.close();
      System.out.println("The First number after swapping:"+num1);
      System.out.println("The Second number after swapping:"+num2);
   }
}
```

Output:

```
Enter first number:10
Enter second number:5
The First number after swapping:5
The Second number after swapping:10
```

Example: Program to find the largest number using ternary operator

```
import java.util.Scanner;

public class JavaExample
{
   public static void main(String[] args)
   {
      int num1, num2, num3, result, temp;

      Scanner scanner = new Scanner(System.in);
      System.out.println("Enter First Number:");
      num1 = scanner.nextInt();
      System.out.println("Enter Second Number:");
      num2 = scanner.nextInt();
      System.out.println("Enter Third Number:");
      num3 = scanner.nextInt();
      scanner.close();

      temp = num1>num2 ? num1:num2;
      result = num3>temp ? num3:temp;
      System.out.println("Largest Number is:"+result);
   }
}
```

Output:

```
Enter First Number:
89
Enter Second Number:
109
Enter Third Number:
8
Largest Number is:109
```

Example: Program to check whether the input year is leap or not

```
import java.util.Scanner;
public class Demo {

   public static void main(String[] args) {

            int year;
            Scanner scan = new Scanner(System.in);
            System.out.println("Enter any Year:");
            year = scan.nextInt();
            scan.close();
      boolean isLeap = false;
```

```
    if(year % 4 == 0)
    {
       if( year % 100 == 0)
       {
          if ( year % 400 == 0)
             isLeap = true;
          else
             isLeap = false;
       }
       else
          isLeap = true;
    }
    else {
       isLeap = false;
    }

    if(isLeap==true)
       System.out.println(year + " is a Leap Year.");
    else
       System.out.println(year + " is not a Leap Year.");
  }
}
```

Output:

```
Enter any Year:
2001
2001 is not a Leap Year.
```

Reverse a number using recursion

```
import java.util.Scanner;
class RecursionReverseDemo
{
  //A method for reverse
  public static void reverseMethod(int number) {
    if (number < 10) {
              System.out.println(number);
              return;
    }
    else {
       System.out.print(number % 10);
       //Method is calling itself: recursion
       reverseMethod(number/10);
    }
  }
  public static void main(String args[])
  {
            int num=0;
            System.out.println("Input your number and press enter: ");
```

```
            Scanner in = new Scanner(System.in);
            num = in.nextInt();
            System.out.print("Reverse of the input number is:");
            reverseMethod(num);
            System.out.println();
  }
}
```

Output:

```
Input your number and press enter:
5678901
Reverse of the input number is:1098765
```

calculate area and circumference of circle

```
class CircleDemo2
{
  public static void main(String args[])
  {
    int radius = 3;
    double area = Math.PI * (radius * radius);
    System.out.println("The area of circle is: " + area);
    double circumference= Math.PI * 2*radius;
    System.out.println( "The circumference of the circle is:"+circumference) ;
  }
}
```

Output:

```
The area of circle is: 28.274333882308138
The circumference of the circle is:18.84955592153876
```

calculate area of Triangle

```
import java.util.Scanner;
class AreaTriangleDemo {
  public static void main(String args[]) {
    Scanner scanner = new Scanner(System.in);

    System.out.println("Enter the width of the Triangle:");
    double base = scanner.nextDouble();

    System.out.println("Enter the height of the Triangle:");
    double height = scanner.nextDouble();

    //Area = (width*height)/2
    double area = (base* height)/2;
    System.out.println("Area of Triangle is: " + area);
  }
```

```
}
```
Output:

```
Enter the width of the Triangle:
2
Enter the height of the Triangle:
2
Area of Triangle is: 2.0
```

sum the elements of an array

```
class SumOfArray{
  public static void main(String args[]){
    int[] array = {10, 20, 30, 40, 50, 10};
    int sum = 0;
    //Advanced for loop
    for( int num : array) {
       sum = sum+num;
    }
    System.out.println("Sum of array elements is:"+sum);
  }
}
```
Output:

```
Sum of array elements is:160
```

Program to check whether input number is prime or not

```
import java.util.Scanner;
class PrimeCheck
{
  public static void main(String args[])
  {
            int temp;
            boolean isPrime=true;
            Scanner scan= new Scanner(System.in);
            System.out.println("Enter any number:");
            //capture the input in an integer
            int num=scan.nextInt();
     scan.close();
            for(int i=2;i<=num/2;i++)
            {
       temp=num%i;
              if(temp==0)
              {
                isPrime=false;
                break;
              }
```

```
            }
            //If isPrime is true then the number is prime else not
            if(isPrime)
              System.out.println(num + " is a Prime Number");
            else
              System.out.println(num + " is not a Prime Number");
  }
}
```

Output:

```
Enter any number:
19
19 is a Prime Number
```

Java Program to check Even or Odd number

```
import java.util.Scanner;

class CheckEvenOdd
{
 public static void main(String args[])
 {
  int num;
  System.out.println("Enter an Integer number:");

  //The input provided by user is stored in num
  Scanner input = new Scanner(System.in);
  num = input.nextInt();

  /* If number is divisible by 2 then it's an even number
   * else odd number*/
  if ( num % 2 == 0 )
     System.out.println("Entered number is even");
   else
     System.out.println("Entered number is odd");
 }
}
```

Output 1:

```
Enter an Integer number:
78
Entered number is even
```

Binary to Decimal conversion using Integer.parseInt() method

```
import java.util.Scanner;
class BinaryToDecimal {
   public static void main(String args[]){
     Scanner input = new Scanner( System.in );
```

```
    System.out.print("Enter a binary number: ");
    String binaryString =input.nextLine();
    System.out.println("Output: "+Integer.parseInt(binaryString,2));
  }
}
```

Output:

```
Enter a binary number: 1101
Output: 13
```

generate random numbers

```
import java.util.*;
class GenerateRandomNumber {
  public static void main(String[] args) {
    int counter;
    Random rnum = new Random();
    /* Below code would generate 5 random numbers
     * between 0 and 200.
     */
    System.out.println("Random Numbers:");
    System.out.println("***************");
    for (counter = 1; counter <= 5; counter++) {
      System.out.println(rnum.nextInt(200));
    }
  }
}
```

Output:

```
Random Numbers:
***************
135
173
5
17
15
```

binary search algorithm

```
import java.util.Scanner;
class BinarySearchExample
{
  public static void main(String args[])
  {
    int counter, num, item, array[], first, last, middle;
    //To capture user input
    Scanner input = new Scanner(System.in);
```

```
    System.out.println("Enter number of elements:");
    num = input.nextInt();

    //Creating array to store the all the numbers
    array = new int[num];

    System.out.println("Enter " + num + " integers");
    //Loop to store each numbers in array
    for (counter = 0; counter < num; counter++)
       array[counter] = input.nextInt();

    System.out.println("Enter the search value:");
    item = input.nextInt();
    first = 0;
    last = num - 1;
    middle = (first + last)/2;

    while( first <= last )
    {
      if ( array[middle] < item )
        first = middle + 1;
      else if ( array[middle] == item )
      {
        System.out.println(item + " found at location " + (middle + 1) + ".");
        break;
      }
      else
      {
         last = middle - 1;
      }
      middle = (first + last)/2;
    }
    if ( first > last )
       System.out.println(item + " is not found.\n");
  }
}
```

Output 1:

```
Enter number of elements:
7
Enter 7 integers
4
5
66
77
8
99
0
Enter the search value:
77
77 found at location 4.
```

linear search algorithm

```
import java.util.Scanner;
class LinearSearchExample
{
  public static void main(String args[])
  {
    int counter, num, item, array[];
    //To capture user input
    Scanner input = new Scanner(System.in);
    System.out.println("Enter number of elements:");
    num = input.nextInt();
    //Creating array to store the all the numbers
    array = new int[num];
    System.out.println("Enter " + num + " integers");
    //Loop to store each numbers in array
    for (counter = 0; counter < num; counter++)
     array[counter] = input.nextInt();

    System.out.println("Enter the search value:");
    item = input.nextInt();

    for (counter = 0; counter < num; counter++)
    {
      if (array[counter] == item)
      {
        System.out.println(item+" is present at location "+(counter+1));
        /*Item is found so to stop the search and to come out of the
         * loop use break statement.*/
        break;
      }
    }
    if (counter == num)
      System.out.println(item + " doesn't exist in array.");
  }
}
```

Output 1:

```
Enter number of elements:
6
Enter 6 integers
22
33
45
1
3
99
Enter the search value:
45
```

```
45 is present at location 3
```

get input from user

```
import java.util.Scanner;

class GetInputData
{
  public static void main(String args[])
  {
    int num;
    float fnum;
    String str;

    Scanner in = new Scanner(System.in);

    //Get input String
    System.out.println("Enter a string: ");
    str = in.nextLine();
    System.out.println("Input String is: "+str);

    //Get input Integer
    System.out.println("Enter an integer: ");
    num = in.nextInt();
    System.out.println("Input Integer is: "+num);

    //Get input float number
    System.out.println("Enter a float number: ");
    fnum = in.nextFloat();
    System.out.println("Input Float number is: "+fnum);
  }
}
```

Output:

```
Enter a string:
Chaitanya
Input String is: Chaitanya
Enter an integer:
27
Input Integer is: 27
Enter a float number:
12.56
Input Float number is: 12.56
```

Program to check whether the given number is Armstrong number

```
public class JavaExample {
```

```
  public static void main(String[] args) {

    int num = 370, number, temp, total = 0;

    number = num;
    while (number != 0)
    {
       temp = number % 10;
       total = total + temp*temp*temp;
       number /= 10;
    }

    if(total == num)
       System.out.println(num + " is an Armstrong number");
    else
       System.out.println(num + " is not an Armstrong number");
  }
}
```

Output:

```
370 is an Armstrong number
```

Example 2: Program to check whether the input number is Armstrong or not

```
import java.util.Scanner;
public class JavaExample {

  public static void main(String[] args) {

    int num, number, temp, total = 0;
    System.out.println("Ënter 3 Digit Number");
    Scanner scanner = new Scanner(System.in);
    num = scanner.nextInt();
    scanner.close();
    number = num;

    for( ;number!=0;number /= 10)
    {
       temp = number % 10;
       total = total + temp*temp*temp;
    }

    if(total == num)
       System.out.println(num + " is an Armstrong number");
    else
       System.out.println(num + " is not an Armstrong number");
  }
}
```

Output:

```
Ënter 3 Digit Number
371
371 is an Armstrong number
```

Program to find the sum of multiple numbers using Method Overloading

```
public class JavaExample
{
   int add(int num1, int num2)
   {
      return num1+num2;
   }
   int add(int num1, int num2, int num3)
   {
      return num1+num2+num3;
   }
   int add(int num1, int num2, int num3, int num4)
   {
      return num1+num2+num3+num4;
   }
   public static void main(String[] args)
   {
            JavaExample obj = new JavaExample();
            //This will call the first add method
      System.out.println("Sum of two numbers: "+obj.add(10, 20));
      //This will call second add method
      System.out.println("Sum of three numbers: "+obj.add(10, 20, 30));
      //This will call third add method
      System.out.println("Sum of four numbers: "+obj.add(1,  2, 3, 4));
   }
}
```

Output:

```
Sum of two numbers: 30
Sum of three numbers: 60
Sum of four numbers: 10
```

Program to display the grade of student

```
import java.util.Scanner;

public class JavaExample
{
   public static void main(String args[])
   {
            /* This program assumes that the student has 6 subjects,
             * thats why I have created the array of size 6. You can
```

```
            * change this as per the requirement.
            */
      int marks[] = new int[6];
      int i;
      float total=0, avg;
      Scanner scanner = new Scanner(System.in);


      for(i=0; i<6; i++) {
        System.out.print("Enter Marks of Subject"+(i+1)+":");
        marks[i] = scanner.nextInt();
        total = total + marks[i];
      }
      scanner.close();
      //Calculating average here
      avg = total/6;
      System.out.print("The student Grade is: ");
      if(avg>=80)
      {
         System.out.print("A");
      }
      else if(avg>=60 && avg<80)
      {
        System.out.print("B");
      }
      else if(avg>=40 && avg<60)
      {
         System.out.print("C");
      }
      else
      {
         System.out.print("D");
      }
   }
}
```

Output:

```
Enter Marks of Subject1:40
Enter Marks of Subject2:80
Enter Marks of Subject3:80
Enter Marks of Subject4:40
Enter Marks of Subject5:60
Enter Marks of Subject6:60
The student Grade is: B
```

Program to make a calculator using switch case in Java

```
import java.util.Scanner;

public class JavaExample {
```

```
  public static void main(String[] args) {

            double num1, num2;
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter first number:");


    num1 = scanner.nextDouble();
    System.out.print("Enter second number:");
    num2 = scanner.nextDouble();

    System.out.print("Enter an operator (+, -, *, /): ");
    char operator = scanner.next().charAt(0);

    scanner.close();
    double output;

    switch(operator)
    {
      case '+':
           output = num1 + num2;
         break;

      case '-':
           output = num1 - num2;
         break;

      case '*':
           output = num1 * num2;
         break;

      case '/':
           output = num1 / num2;
         break;

       /* If user enters any other operator or char apart from
        * +, -, * and /, then display an error message to user
        *
        */
       default:
         System.out.printf("You have entered wrong operator");
         return;
    }

    System.out.println(num1+" "+operator+" "+num2+": "+output);
  }
}
```

Output:

```
Enter first number:40
```

```
Enter second number:4
Enter an operator (+, -, *, /): /
40.0 / 4.0: 10.0
```

Program to display the prime numbers from 1 to 100

```
class PrimeNumbers
{
  public static void main (String[] args)
  {
     int i =0;
     int num =0;
     //Empty String
     String  primeNumbers = "";

     for (i = 1; i <= 100; i++)
     {
       int counter=0;
       for(num =i; num>=1; num--)
              {
         if(i%num==0)
                {
                        counter = counter + 1;
                }
              }
              if (counter ==2)
              {
                //Appended the Prime number to the String
                primeNumbers = primeNumbers + i + " ";
              }
     }
     System.out.println("Prime numbers from 1 to 100 are :");
     System.out.println(primeNumbers);
  }
}
```

Output:

```
Prime numbers from 1 to 100 are :
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

Program to display prime numbers from 1 to n

```
import java.util.Scanner;
class PrimeNumbers2
{
  public static void main (String[] args)
  {
    Scanner scanner = new Scanner(System.in);
    int i =0;
    int num =0;
    //Empty String
```

```
    String  primeNumbers = "";
    System.out.println("Enter the value of n:");
    int n = scanner.nextInt();
    scanner.close();
    for (i = 1; i <= n; i++)
    {
      int counter=0;
      for(num =i; num>=1; num--)
      {
                if(i%num==0)
                {
                          counter = counter + 1;
                }
              }
              if (counter ==2)
              {
                //Appended the Prime number to the String
                primeNumbers = primeNumbers + i + " ";
              }
    }
    System.out.println("Prime numbers from 1 to n are :");
    System.out.println(primeNumbers);
  }
}
```

Output:

```
Enter the value of n:
150
Prime numbers from 1 to n are :
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89
97 101 103 107 109 113 127 131 137 139 149
```

Program to check whether the given number is positive or negative

```
public class Demo
{
   public static void main(String[] args)
   {
      int number=109;
      if(number > 0)
      {
         System.out.println(number+" is a positive number");
      }
      else if(number < 0)
      {
         System.out.println(number+" is a negative number");
      }
      else
```

```
    {
        System.out.println(number+" is neither positive nor negative");
    }
  }
}
```

Output:

```
109 is a positive number
```

Example 2: Check whether the input number(entered by user) is positive or negative

```
import java.util.Scanner;
public class Demo
{
  public static void main(String[] args)
  {
    int number;
    Scanner scan = new Scanner(System.in);
    System.out.print("Enter the number you want to check:");
    number = scan.nextInt();
    scan.close();
    if(number > 0)
    {
      System.out.println(number+" is positive number");
    }
    else if(number < 0)
    {
      System.out.println(number+" is negative number");
    }
    else
    {
      System.out.println(number+" is neither positive nor negative");
    }
  }
}
```

Output:

```
Enter the number you want to check:-12
-12 is negative number
```

Program to find ASCII code of a character

```
public class Demo {

  public static void main(String[] args) {

    char ch = 'P';
    int asciiCode = ch;
    // type casting char as int
```

```
      int asciiValue = (int)ch;

      System.out.println("ASCII value of "+ch+" is: " + asciiCode);
      System.out.println("ASCII value of "+ch+" is: " + asciiValue);
   }
}
```

Output:

```
ASCII value of P is: 80
ASCII value of P is: 80
```

IP address

```
import java.net.InetAddress;

class GetMyIPAddress
{
  public static void main(String args[]) throws Exception
  {
    InetAddress myIP=InetAddress.getLocalHost();
    System.out.println("My IP Address is:");
    System.out.println(myIP.getHostAddress());
  }
}
```

Output:

```
My IP Address is:
115.242.7.243
```

Palindrome check Using Stack

```
import java.util.Stack;
import java.util.Scanner;
class PalindromeTest {

   public static void main(String[] args) {

            System.out.print("Enter any string:");
      Scanner in=new Scanner(System.in);
      String inputString = in.nextLine();
      Stack stack = new Stack();

      for (int i = 0; i < inputString.length(); i++) {
         stack.push(inputString.charAt(i));
      }

      String reverseString = "";
```

```
      while (!stack.isEmpty()) {
         reverseString = reverseString+stack.pop();
      }

      if (inputString.equals(reverseString))
         System.out.println("The input String is a palindrome.");
      else
         System.out.println("The input String is not a palindrome.");

   }
}
```

Output 1:

```
Enter any string:abccba
The input String is a palindrome.
```

Output 2:

```
Enter any string:abcdef
The input String is not a palindrome.
```

Program 2: Palindrome check Using Queue

```
import java.util.Queue;
import java.util.Scanner;
import java.util.LinkedList;
class PalindromeTest {

   public static void main(String[] args) {

            System.out.print("Enter any string:");
      Scanner in=new Scanner(System.in);
      String inputString = in.nextLine();
      Queue queue = new LinkedList();

      for (int i = inputString.length()-1; i >=0; i--) {
         queue.add(inputString.charAt(i));
      }

      String reverseString = "";

      while (!queue.isEmpty()) {
         reverseString = reverseString+queue.remove();
      }
      if (inputString.equals(reverseString))
         System.out.println("The input String is a palindrome.");
      else
         System.out.println("The input String is not a palindrome.");

   }
}
```

Output 1:

```
Enter any string:xyzzyx
xyzzyx
The input String is a palindrome.
```

Output 2:

```
Enter any string:xyz
The input String is not a palindrome.
```

Program 3: Using for loop/While loop and String function charAt

```
import java.util.Scanner;
class PalindromeTest {
  public static void main(String args[])
  {
    String reverseString="";
    Scanner scanner = new Scanner(System.in);

    System.out.println("Enter a string to check if it is a palindrome:");
    String inputString = scanner.nextLine();

    int length = inputString.length();

    for ( int i = length - 1 ; i >= 0 ; i-- )
      reverseString = reverseString + inputString.charAt(i);

    if (inputString.equals(reverseString))
      System.out.println("Input string is a palindrome.");
    else
      System.out.println("Input string is not a palindrome.");

  }
}
```

Output 1:

```
Enter a string to check if it is a palindrome:
aabbaa
Input string is a palindrome.
```

Output 2:

```
Enter a string to check if it is a palindrome:
aaabbb
Input string is not a palindrome.
```

Projects

Calculator::::::::::::

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  /*********************************************/
4.
5.  public class MyCalculator extends Frame
6.  {
7.
8.  public boolean setClear=true;
9.  double number, memValue;
10. char op;
11.
12. String digitButtonText[] = {"7", "8", "9", "4", "5", "6", "1", "2", "3", "0", "+/-", "." };
13. String operatorButtonText[] = {"/", "sqrt", "*", "%", "-", "1/X", "+", "=" };
14. String memoryButtonText[] = {"MC", "MR", "MS", "M+" };
15. String specialButtonText[] = {"Backspc", "C", "CE" };
16.
17. MyDigitButton digitButton[]=new MyDigitButton[digitButtonText.length];
18. MyOperatorButton operatorButton[]=new MyOperatorButton[operatorButtonText.length];
19. MyMemoryButton memoryButton[]=new MyMemoryButton[memoryButtonText.length];
20. MySpecialButton specialButton[]=new MySpecialButton[specialButtonText.length];
21.
22. Label displayLabel=new Label("0",Label.RIGHT);
23. Label memLabel=new Label(" ",Label.RIGHT);
24.
25. final int FRAME_WIDTH=325,FRAME_HEIGHT=325;
26. final int HEIGHT=30, WIDTH=30, H_SPACE=10,V_SPACE=10;
27. final int TOPX=30, TOPY=50;
```

```
28. //////////////////////////
29. MyCalculator(String frameText)//constructor
30. {
31. super(frameText);
32. 
33. int tempX=TOPX, y=TOPY;
34. displayLabel.setBounds(tempX,y,240,HEIGHT);
35. displayLabel.setBackground(Color.BLUE);
36. displayLabel.setForeground(Color.WHITE);
37. add(displayLabel);
38. 
39. memLabel.setBounds(TOPX,  TOPY+HEIGHT+ V_SPACE,WIDTH, HEIGHT);
40. add(memLabel);
41. 
42. // set Co-ordinates for Memory Buttons
43. tempX=TOPX;
44. y=TOPY+2*(HEIGHT+V_SPACE);
45. for(int i=0; i<memoryButton.length; i++)
46. {
47. memoryButton[i]=new MyMemoryButton(tempX,y,WIDTH,HEIGHT,memoryButtonText[i], this);
48. memoryButton[i].setForeground(Color.RED);
49. y+=HEIGHT+V_SPACE;
50. }
51. 
52. //set Co-ordinates for Special Buttons
53. tempX=TOPX+1*(WIDTH+H_SPACE); y=TOPY+1*(HEIGHT+V_SPACE);
54. for(int i=0;i<specialButton.length;i++)
55. {
56. specialButton[i]=new MySpecialButton(tempX,y,WIDTH*2,HEIGHT,specialButtonText[i], this);
57. specialButton[i].setForeground(Color.RED);
58. tempX=tempX+2*WIDTH+H_SPACE;
59. }
60. 
61. //set Co-ordinates for Digit Buttons
62. int digitX=TOPX+WIDTH+H_SPACE;
63. int digitY=TOPY+2*(HEIGHT+V_SPACE);
64. tempX=digitX;  y=digitY;
65. for(int i=0;i<digitButton.length;i++)
66. {
67. digitButton[i]=new MyDigitButton(tempX,y,WIDTH,HEIGHT,digitButtonText[i], this);
68. digitButton[i].setForeground(Color.BLUE);
69. tempX+=WIDTH+H_SPACE;
70. if((i+1)%3==0){tempX=digitX; y+=HEIGHT+V_SPACE;}
71. }
72. 
73. //set Co-ordinates for Operator Buttons
74. int opsX=digitX+2*(WIDTH+H_SPACE)+H_SPACE;
75. int opsY=digitY;
```

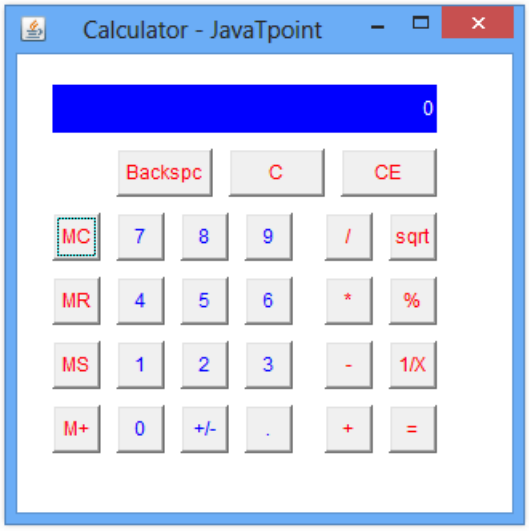
```
76. tempX=opsX;  y=opsY;
77. for(int i=0;i<operatorButton.length;i++)
78. {
79. tempX+=WIDTH+H_SPACE;
80. operatorButton[i]=new MyOperatorButton(tempX,y,WIDTH,HEIGHT,operatorButtonText[i], this);
81. operatorButton[i].setForeground(Color.RED);
82. if((i+1)%2==0){tempX=opsX; y+=HEIGHT+V_SPACE;}
83. }
84.
85. addWindowListener(new WindowAdapter()
86. {
87. public void windowClosing(WindowEvent ev)
88. {System.exit(0);}
89. });
90.
91. setLayout(null);
92. setSize(FRAME_WIDTH,FRAME_HEIGHT);
93. setVisible(true);
94. }
95. ////////////////////////////////
96. static String getFormattedText(double temp)
97. {
98. String resText=""+temp;
99. if(resText.lastIndexOf(".0")>0)
100.             resText=resText.substring(0,resText.length()-2);
101.         return resText;
102.         }
103.         ////////////////////////////////////////
104.         public static void main(String []args)
105.         {
106.         new MyCalculator("Calculator - JavaTpoint");
107.         }
108.         }
109.
110.         /*****************************************/
111.
112.         class MyDigitButton extends Button implements ActionListener
113.         {
114.         MyCalculator cl;
115.
116.         ////////////////////////////////////////
117.         MyDigitButton(int x,int y, int width,int height,String cap, MyCalculator clc)
118.         {
119.         super(cap);
120.         setBounds(x,y,width,height);
121.         this.cl=clc;
122.         this.cl.add(this);
123.         addActionListener(this);
```

```
124.        }
125.        ////////////////////////////////////////////
126.        static boolean isInString(String s, char ch)
127.        {
128.        for(int i=0; i<s.length();i++) if(s.charAt(i)==ch) return true;
129.        return false;
130.        }
131.        ////////////////////////////////////////////
132.        public void actionPerformed(ActionEvent ev)
133.        {
134.        String tempText=((MyDigitButton)ev.getSource()).getLabel();
135.
136.        if(tempText.equals("."))
137.        {
138.         if(cl.setClear)
139.           {cl.displayLabel.setText("0.");cl.setClear=false;}
140.         else if(!isInString(cl.displayLabel.getText(),'.'))
141.           cl.displayLabel.setText(cl.displayLabel.getText()+".");
142.         return;
143.        }
144.
145.        int index=0;
146.        try{
147.              index=Integer.parseInt(tempText);
148.           }catch(NumberFormatException e){return;}
149.
150.        if (index==0 && cl.displayLabel.getText().equals("0")) return;
151.
152.        if(cl.setClear)
153.                {cl.displayLabel.setText(""+index);cl.setClear=false;}
154.        else
155.           cl.displayLabel.setText(cl.displayLabel.getText()+index);
156.        }//actionPerformed
157.        }//class defination
158.
159.        /*****************************************/
160.
161.        class MyOperatorButton extends Button implements ActionListener
162.        {
163.        MyCalculator cl;
164.
165.        MyOperatorButton(int x,int y, int width,int height,String cap, MyCalculator clc)
166.        {
167.        super(cap);
168.        setBounds(x,y,width,height);
169.        this.cl=clc;
170.        this.cl.add(this);
171.        addActionListener(this);
```

```
172.        }
173.        //////////////////////
174.        public void actionPerformed(ActionEvent ev)
175.        {
176.        String opText=((MyOperatorButton)ev.getSource()).getLabel();
177.
178.        cl.setClear=true;
179.        double temp=Double.parseDouble(cl.displayLabel.getText());
180.
181.        if(opText.equals("1/x"))
182.           {
183.           try
184.              {double tempd=1/(double)temp;
185.              cl.displayLabel.setText(MyCalculator.getFormattedText(tempd));}
186.           catch(ArithmeticException excp)
187.                        {cl.displayLabel.setText("Divide by 0.");}
188.           return;
189.           }
190.        if(opText.equals("sqrt"))
191.           {
192.           try
193.              {double tempd=Math.sqrt(temp);
194.              cl.displayLabel.setText(MyCalculator.getFormattedText(tempd));}
195.                 catch(ArithmeticException excp)
196.                       {cl.displayLabel.setText("Divide by 0.");}
197.           return;
198.           }
199.        if(!opText.equals("="))
200.           {
201.           cl.number=temp;
202.           cl.op=opText.charAt(0);
203.           return;
204.           }
205.        // process = button pressed
206.        switch(cl.op)
207.        {
208.        case '+':
209.           temp+=cl.number;break;
210.        case '-':
211.           temp=cl.number-temp;break;
212.        case '*':
213.           temp*=cl.number;break;
214.        case '%':
215.           try{temp=cl.number%temp;}
216.           catch(ArithmeticException excp)
217.              {cl.displayLabel.setText("Divide by 0."); return;}
218.           break;
219.        case '/':
```

```
220.            try{temp=cl.number/temp;}
221.               catch(ArithmeticException excp)
222.                     {cl.displayLabel.setText("Divide by 0."); return;}
223.            break;
224.         }//switch
225.
226.         cl.displayLabel.setText(MyCalculator.getFormattedText(temp));
227.         //cl.number=temp;
228.         }//actionPerformed
229.         }//class
230.
231.         /*****************************************/
232.
233.         class MyMemoryButton extends Button implements ActionListener
234.         {
235.         MyCalculator cl;
236.
237.         ////////////////////////////////
238.         MyMemoryButton(int x,int y, int width,int height,String cap, MyCalculator clc)
239.         {
240.         super(cap);
241.         setBounds(x,y,width,height);
242.         this.cl=clc;
243.         this.cl.add(this);
244.         addActionListener(this);
245.         }
246.         ////////////////////////////////////////////
247.         public void actionPerformed(ActionEvent ev)
248.         {
249.         char memop=((MyMemoryButton)ev.getSource()).getLabel().charAt(1);
250.
251.         cl.setClear=true;
252.         double temp=Double.parseDouble(cl.displayLabel.getText());
253.
254.         switch(memop)
255.         {
256.         case 'C':
257.            cl.memLabel.setText(" ");cl.memValue=0.0;break;
258.         case 'R':
259.            cl.displayLabel.setText(MyCalculator.getFormattedText(cl.memValue));break;
260.         case 'S':
261.            cl.memValue=0.0;
262.         case '+':
263.            cl.memValue+=Double.parseDouble(cl.displayLabel.getText());
264.            if(cl.displayLabel.getText().equals("0") || cl.displayLabel.getText().equals("0.0")  )
265.               cl.memLabel.setText(" ");
266.            else
267.               cl.memLabel.setText("M");
```

```
268.            break;
269.         }//switch
270.         }//actionPerformed
271.         }//class
272.
273.         /*****************************************/
274.
275.         class MySpecialButton extends Button implements Ac
     tionListener
276.         {
277.         MyCalculator cl;
278.
279.         MySpecialButton(int x,int y, int width,int height,String
      cap, MyCalculator clc)
280.         {
281.         super(cap);
282.         setBounds(x,y,width,height);
283.         this.cl=clc;
284.         this.cl.add(this);
285.         addActionListener(this);
286.         }
287.         //////////////////////
288.         static String backSpace(String s)
289.         {
290.         String Res="";
291.         for(int i=0; i<s.length()-1; i++) Res+=s.charAt(i);
292.         return Res;
293.         }
294.
295.         ////////////////////////////////////////////////////////
296.         public void actionPerformed(ActionEvent ev)
297.         {
298.         String opText=((MySpecialButton)ev.getSource()).getLabel();
299.         //check for backspace button
300.         if(opText.equals("Backspc"))
301.         {
302.         String tempText=backSpace(cl.displayLabel.getText());
303.         if(tempText.equals(""))
304.            cl.displayLabel.setText("0");
305.         else
306.            cl.displayLabel.setText(tempText);
307.         return;
308.         }
309.         //check for "C" button i.e. Reset
310.         if(opText.equals("C"))
311.         {
312.         cl.number=0.0; cl.op=' '; cl.memValue=0.0;
313.         cl.memLabel.setText(" ");
```



```
314.          }
315.
316.          //it must be CE button pressed
317.          cl.displayLabel.setText("0");cl.setClear=true;
318.          }//actionPerformed
319.          }//class
```

URL Source Code Generator

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.io.InputStream;
4.  import java.net.*;
5.  public class SourceGetter extends Frame implements ActionListener{
6.     TextField tf;
7.     TextArea ta;
8.     Button b;
9.     Label l;
10.    SourceGetter(){
11.       super("Source Getter Tool - Javatpoint");
12.       l=new Label("Enter URL:");
13.       l.setBounds(50,50,50,20);
14.
15.       tf=new TextField();
16.       tf.setBounds(120,50,250,20);
17.
18.       b=new Button("Get Source Code");
19.       b.setBounds(120, 100,120,30);
20.       b.addActionListener(this);
21.
22.       ta=new TextArea();
23.       ta.setBounds(120,150,250,150);
24.
25.       add(l);add(tf);add(b);add(ta);
26.       setSize(400,400);
27.       setLayout(null);
28.       setVisible(true);
29.    }
30.    public void actionPerformed(ActionEvent e){
31.       String s=tf.getText();
32.       if(s==null){}
33.       else{
34.          try{
35.          URL u=new URL(s);
36.          URLConnection uc=u.openConnection();
37.
38.          InputStream is=uc.getInputStream();
39.          int i;
```

```
40.          StringBuilder sb=new StringBuilder();
41.          while((i=is.read())!=-1){
42.             sb.append((char)i);
43.          }
44.          String source=sb.toString();
45.          ta.setText(source);
46.          }catch(Exception ex){System.out.println(e);}
47.       }
48.    }
49.    public static void main(String[] args) {
50.       new SourceGetter();
51.    }
52. }
```

******

```
1.  URL u=new URL("https://www.facebook.com");//change the URL
2.  URLConnection uc=u.openConnection();
3.  InputStream is=uc.getInputStream();
4.  int i;
5.  StringBuilder sb=new StringBuilder();
6.  while((i=is.read())!=-1){
7.     sb.append((char)i);
8.  }
9.  String source=sb.toString();
```

Snake games:

```
/**
         *
         * @author pau
         */
         public class Main {
            /**
             * @param args the command line arguments
             */
            public static void main(String[] args) {
               new Snake();
            }
         }
```

****__***

```
import
java.awt.*;
                  import java.awt.event.*;
                  import javax.swing.*;
                  import java.util.*;
                  class Snake extends JFrame implements KeyListener, Runnable {
                     JPanel p1, p2;
                     JButton[] lb = new JButton[200];
                     JButton bonusfood;
                     JTextArea t;
                     int x = 500, y = 250, gu = 2, directionx = 1, directiony = 0, speed = 50, difference = 0,
```

```
oldx, oldy, score = 0;
   int[] lbx = new int[300];
   int[] lby = new int[300];
   Point[] lbp = new Point[300];
   Point bfp = new Point();
   Thread myt;
   boolean food = false, runl = false, runr = true, runu = true, rund = true, bonusflag =
true;
   Random r = new Random();
   JMenuBar mymbar;
   JMenu game, help, level;
   public void initializeValues() {
      gu = 3;
      lbx[0] = 100;
      lby[0] = 150;
      directionx = 10;
      directiony = 0;
      difference = 0;
      score = 0;
      food = false;
      runl = false;
      runr = true;
      runu = true;
      rund = true;
      bonusflag = true;
   }
   Snake() {
      super("Snake");
      setSize(500, 330);
      //Create Menue bar with functions
      creatbar();
      //initialize all variables
      initializeValues();
      // Start of UI design
      p1 = new JPanel();
      p2 = new JPanel();
      // t will view the score
      t = new JTextArea("Score ==>" + score);
      t.setEnabled(false);
      t.setBackground(Color.BLACK);
      // snake have to eat bonousfood to growup
      bonusfood = new JButton();
      bonusfood.setEnabled(false);
      // will make first snake
      createFirstSnake();
      p1.setLayout(null);
      p2.setLayout(new GridLayout(0, 1));
      p1.setBounds(0, 0, x, y);
      p1.setBackground(Color.blue);
      p2.setBounds(0, y, x, 30);
      p2.setBackground(Color.RED);
```

```
    p2.add(t); // will contain score board
    // end of UI design
    getContentPane().setLayout(null);
    getContentPane().add(p1);
    getContentPane().add(p2);
    show();
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    addKeyListener(this);
    // start thread
    myt = new Thread(this);
    myt.start(); // go to run() method
}
public void createFirstSnake() {
    // Initially the snake has small length 3
    for (int i = 0; i < 3; i++) {
        lb[i] = new JButton("lb" + i);
        lb[i].setEnabled(false);
        p1.add(lb[i]);
        lb[i].setBounds(lbx[i], lby[i], 10, 10);
        lbx[i + 1] = lbx[i] - 10;
        lby[i + 1] = lby[i];
    }
}
public void creatbar() {
    mymbar = new JMenuBar();
    game = new JMenu("Game");
    JMenuItem newgame = new JMenuItem("New Game");
    JMenuItem exit = new JMenuItem("Exit");
    newgame.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    reset();
                }
            });
    exit.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            System.exit(0);
        }
    });
    game.add(newgame);
    game.addSeparator();
    game.add(exit);
    mymbar.add(game);
    level = new JMenu("Level");
    mymbar.add(level);
    help = new JMenu("Help");
    JMenuItem creator = new JMenuItem("Creator");
    JMenuItem instruction = new JMenuItem("Instraction");
    creator.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(p2, "Name :md.murad\nRollno
```

```
:161060042\nSub :cse\ninstitute :pau");
            }
        });
        help.add(creator);
        help.add(instruction);
        mymbar.add(help);
        setJMenuBar(mymbar);
    }
    void reset() {
        initializeValues();
        p1.removeAll();
        myt.stop();
        createFirstSnake();
        t.setText("Score==>" + score);
        myt = new Thread(this);
        myt.start();
    }
    void growup() {
        lb[gu] = new JButton();
        lb[gu].setEnabled(false);
        p1.add(lb[gu]);
        int a = 10 + (10 * r.nextInt(48));
        int b = 10 + (10 * r.nextInt(23));
        lbx[gu] = a;
        lby[gu] = b;
        lb[gu].setBounds(a, b, 10, 10);
        gu++;
    }
    // this method contains the logic to move the snake. player will define the derection
    // this method just forward the snake to the right derection which deriction is
pressed
    // by the player.
    void moveForward() {
        for (int i = 0; i < gu; i++) {
            lbp[i] = lb[i].getLocation();
        }
        lbx[0] += directionx;
        lby[0] += directiony;
        lb[0].setBounds(lbx[0], lby[0], 10, 10);
        for (int i = 1; i < gu; i++) {
            lb[i].setLocation(lbp[i - 1]);
        }
        if (lbx[0] == x) {
            lbx[0] = 10;
        } else if (lbx[0] == 0) {
            lbx[0] = x - 10;
        } else if (lby[0] == y) {
            lby[0] = 10;
        } else if (lby[0] == 0) {
            lby[0] = y - 10;
        }
```

```
      if (lbx[0] == lbx[gu - 1] && lby[0] == lby[gu - 1]) {
         food = false;
         score += 5;
         t.setText("Score==>" + score);
         if (score % 50 == 0 && bonusflag == true) {
            p1.add(bonusfood);
            bonusfood.setBounds((10 * r.nextInt(50)), (10 * r.nextInt(25)), 15, 15);
            bfp = bonusfood.getLocation();
            bonusflag = false;
         }
      }
      if (bonusflag == false) {
         if (bfp.x <= lbx[0] && bfp.y <= lby[0] && bfp.x + 10 >= lbx[0] && bfp.y + 10 >=
lby[0]) {
            p1.remove(bonusfood);
            score += 100;
            t.setText("Score ==>" + score);
            bonusflag = true;
         }
      }
      if (food == false) {
         growup();
         food = true;
      } else {
         lb[gu - 1].setBounds(lbx[gu - 1], lby[gu - 1], 10, 10);
      }
      for (int i = 1; i < gu; i++) {
         if (lbp[0] == lbp[i]) {
            t.setText("GAME OVER " + score);
            try {
               myt.join();
            } catch (InterruptedException ie) {
            }
            break;
         }
      }
      p1.repaint();
      show();
   }
   public void keyPressed(KeyEvent e) {
      // snake move to left when player pressed left arrow
      if (runl == true && e.getKeyCode() == 37) {
         directionx = -10; // means snake move right to left by 10pixels
         directiony = 0;
         runr = false;    // run right(runr) means snake cant move from left to right
         runu = true;     // run up   (runu) means snake can move from down to up
         rund = true;     // run down (run down) means snake can move from up to down
      }
      // snake move to up when player pressed up arrow
      if (runu == true && e.getKeyCode() == 38) {
         directionx = 0;
```

```
            directiony = -10; // means snake move from down to up by 10 pixel
            rund = false;     // run down (run down) means snake can move from up to down
            runr = true;      // run right(runr) means snake can move from left to right
            runl = true;      // run left (runl) means snake can move from right to left
         }
         // snake move to right when player pressed right arrow
         if (runr == true && e.getKeyCode() == 39) {
            directionx = +10; // means snake move from left to right by 10 pixel
            directiony = 0;
            runl = false;
            runu = true;
            rund = true;
         }
         // snake move to down when player pressed down arrow
         if (rund == true && e.getKeyCode() == 40) {
            directionx = 0;
            directiony = +10; // means snake move from left to right by 10 pixel
            runu = false;
            runr = true;
            runl = true;
         }
      }
      public void keyReleased(KeyEvent e) {
      }
      public void keyTyped(KeyEvent e) {
      }
      public void run() {
         for (;;) {
            // Move the snake move forword. In the start of the game snake move left to
right,
            // if player press up, down, right or left arrow snake change its direction
according to
            // pressed arrow
            moveForward();
            try {
               Thread.sleep(speed);
            } catch (InterruptedException ie) {
            }
         }
      }
   }
```

Digital clock:

```
package
digitalclock;
                import java.awt.Font;
                import java.awt.Color;
                import java.awt.GridLayout;
```

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.Timer;
import javax.swing.SwingConstants;
import java.util.*;
import java.text.*;
public class DigitalClock {
public static void main(String[] arguments) {

   ClockLabel dateLable = new ClockLabel("date");
   ClockLabel timeLable = new ClockLabel("time");
   ClockLabel dayLable = new ClockLabel("day");

   JFrame.setDefaultLookAndFeelDecorated(true);
   JFrame f = new JFrame("Digital Clock");
   f.setSize(300,150);
   f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   f.setLayout(new GridLayout(3, 1));
   f.add(dateLable);
   f.add(timeLable);
   f.add(dayLable);
   f.getContentPane().setBackground(Color.black);
   f.setVisible(true);
  }
}
class ClockLabel extends JLabel implements ActionListener {
  String type;
  SimpleDateFormat sdf;
  ublic ClockLabel(String type) {
   this.type = type;
  //setForeground(Color.yellow) ;//
// all kind of colors
   switch (type) {
     case "date" : sdf = new SimpleDateFormat("  MMMM dd yyyy");
                   setFont(new Font("sans-serif", Font.PLAIN, 12));
                   setHorizontalAlignment(SwingConstants.LEFT);
                   setForeground(Color.yellow);
                   break;
     case "time" : sdf = new SimpleDateFormat("hh:mm:ss a");
                   setFont(new Font("sans-serif", Font.PLAIN, 40));
                   setHorizontalAlignment(SwingConstants.CENTER);
                   setForeground(Color.green);
                   break;
     case "day"  : sdf = new SimpleDateFormat("EEEE  ");
                   setFont(new Font("sans-serif", Font.PLAIN, 16));
                   setHorizontalAlignment(SwingConstants.RIGHT);
                   setForeground(Color.white);
                   break;
     default     : sdf = new SimpleDateFormat();
```

```
                    break;
                }
                Timer t = new Timer(1000, this);
                t.start();
            }
            public void actionPerformed(ActionEvent ae) {
                Date d = new Date();
                setText(sdf.format(d));
            }
        }
```

Analog clock:

```
package
clock;
        import java.awt.*;
        import java.awt.event.*;
        import javax.swing.*;
        import java.util.*;
        //Class Definition: clock
        public class Clock  extends Frame implements MouseMotionListener{
        int xmou=200; //set the center of circle
        int ymou=200; //set the center of circle
        double theta=-0.1047; //theta for second's hand
        int x=xmou;  //x position of Second's hand
        int y=ymou;  //y position of second's hand
        int p,b;   //perpendicular and base of Second's hand
        int h;   //hypotenous(heigth) of clock's hand
        double the= -0.1047;  //theta for creating outer circle
        double thetamin=-0.1047;       //theta for minutes hand
        int xm=xmou;   //x position of minute's hand
        int ym=ymou;   //y position of minute's hand
        int pmin,bmin;   //perpendicular and base of Minute's hand
        double thetah=-0.1047;       //theta for hour hand
        int xh=xmou;   //y position of hour's hand
        int yh=ymou;   //y position of hour's hand
        int ph,bh;   //perpendicular and base of hour's hand
        double thetan=-0.0;        //theta for numbers of clock
        int xn=xmou;   //x position of Clock numbers
        int yn=ymou;   //y position of Clock numbers
        int pn,bn;   //perpendicular and base of clock numbers
        int num=0;   //for writing the numbers
        //constructor
        Clock(){
```

```
super();
setSize(500,500);
setBackground(Color.WHITE);
setVisible(true);
addMouseMotionListener(this);
}
//method of implemented  mouse interface
public void mouseMoved(MouseEvent me){
}
public void mouseDragged(MouseEvent me){
xmou=me.getX();   //changing the clock position on mouse drag
ymou=me.getY();   //changing the clock position on mouse drag
}
//method to paint clock
public void paint(Graphics g){

        //for writing numbers in clock and outer circle
 for(int p=0;p<60;p++){

       int xocir=xmou;      //x position of outer circle
    int yocir=ymou;      //y position of outer circle
    int pocir,bocir;     //perpendicular and base of outer circle
         pocir=  (int) (Math.sin(the) * (h+23));
         bocir=  (int) (Math.cos(the) * (h+23));
         xocir=xocir-pocir;
         yocir=yocir-bocir;
         the=the - 0.1047;
             g.setColor(Color.RED);
       g.drawLine(xocir+5,yocir+5,xocir,yocir);
           g.setColor(Color.BLACK);
if(p%5==0 ){
 num++;
 if(num>12){
  num=1;
 }

 xn=xmou;
     yn=ymou;

 if(thetan<=-6.28318531 ){
  thetan=0.0;
 }
 thetan=thetan-0.523598776 ;
    pn=  (int) (Math.sin(thetan) * (h+10));
    bn=  (int) (Math.cos(thetan) * (h+10));
    xn=xn-pn;
     yn=yn-bn;
 g.drawString(""+num,xn-3,yn+5);
  }
   }
```

```
      //for drawing Clock hands
    g.setColor(Color.BLACK);

    g.drawLine(xmou,ymou,xm,ym); //drawing minute's hand
    g.drawLine(xmou,ymou,xh,yh); //drawing hour's hand

    g.setColor(Color.RED);
    g.drawLine(xmou,ymou,x,y);  //drawing second's hand

   }
 public void newpoint(){
   Calendar now = Calendar.getInstance(); //creating a Calendar variable for getting
current time
   //for second hand
   x=xmou;
   y=ymou;
   theta=-0.1047;
   theta=theta*now.get(Calendar.SECOND);
   p=  (int) (Math.sin(theta) * h);
   b=  (int) (Math.cos(theta) * h);
   x=x-p;
   y=y-b;
   xm=xmou;
   ym=ymou;
   thetamin=-0.1047;
   thetamin=thetamin*now.get(Calendar.MINUTE);
   pmin=  (int) (Math.sin(thetamin) * (h-6));
   bmin=  (int) (Math.cos(thetamin) * (h-6));
   xm=xm-pmin;
   ym=ym-bmin;
   //for hour's hand
   xh=xmou;
   yh=ymou;
   thetah=-0.1047;
   thetah=thetah*now.get(Calendar.HOUR)*5;
  if (now.get(Calendar.MINUTE)>=12 && now.get(Calendar.MINUTE)<24){
 thetah=thetah-0.1047;
  }
  else if(now.get(Calendar.MINUTE)>=24 && now.get(Calendar.MINUTE)<36){
 thetah=thetah-(2*0.1047);
  }
  else if(now.get(Calendar.MINUTE)>=36 && now.get(Calendar.MINUTE)<48){
 thetah=thetah-(3*0.1047);
  }
  else if(now.get(Calendar.MINUTE)>=48 && now.get(Calendar.MINUTE)<60){
 thetah=thetah-(4*0.1047);
  }
   ph=  (int) (Math.sin(thetah) * (h-15));
   bh=  (int) (Math.cos(thetah) * (h-15));
   xh=xh-ph;
   yh=yh-bh;
```

```
 }
   public static void main(String[] args) {
       Clock m=new Clock();
       m.h=60;
        while(true){
        m.newpoint();
        m.repaint();
        try{
     Thread.sleep(6);
     }catch(Exception e){

     }
      }
    }
}
```

Refarance Books and Web Sites

Books:

- **Effective Java** **by** Joshua Bloch.
- **Clean Code** **by** Robert C. Martin.
- **Java: A Beginner's Guide** **by** Herbert Schildt.
- **JDBC API Tutorial and Reference (3rd Edition)** **by** Maydene Fisher.
- **Learn Java in One Day and Learn It Well.**

“If you are willing to learn, no one can help you. If you are determined to learn, no one can stop you.”